REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
05-10-2013		

4. TITLE AND SUBTITLE

An Algorithm to Identify and Localize Suitable Dock Locations from 3-D LiDAR Scans

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Graves, Mitchell Robert

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Academy
Annapolis, MD 21402

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
Trident Scholar Report no. 415 (2013)

12. DISTRIBUTION / AVAILABILITY STATEMENT

This document has been approved for public release; its distribution is UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Unmanned vehicles have established an important place in the modern battlefield. They play a key role in intelligence and surveillance while not putting human lives in harm's way. A necessary enabling technology is the ability to automatically identify Launch and Recovery sites from sensor data. This project focuses on the identification of suitable docking sites from three dimensional Light Detection and Ranging (LiDAR) scans. A LiDAR sensor is a sensor that collects range images from a rotating array of vertically aligned lasers. Our solution leverages open source C++ code from Point Cloud Library -- "a standalone, large scale, open project for 2D/3D image and point cloud processing." First the Random Sample Consensus (RANSAC) algorithm is used to isolate horizontal planar surfaces that may belong to the dock. After removing planar dock points, Euclidean Cluster Recognition is used to isolate point clusters that are potential vertical pilings. Bayes' Theorem is used to compute the probability that each cluster matches the characteristics of piling. For each candidate piling, the origin of that cluster is compared to the location of the target dock's planar surface. The dock can be identified by the relation of the pilings location to the dock's planar surface. The final output of the algorithm will be a sub-set of points, isolated from the original cloud, that are hypothesized to correspond to the dock.

15. SUBJECT TERMS
Algorithm, Dock, Locations, Point Clouds, LiDAR, Identify

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		87	**19b. TELEPHONE NUMBER** *(include area code)*

U.S.N.A. --- Trident Scholar project report; no. 415 (2013)

An Algorithm to Identify and Localize Suitable Dock Locations from 3-D LiDAR Scans

by

Midshipman 1/c Mitchell R. Graves
United States Naval Academy
Annapolis, Maryland

_____
(signature)

Certification of Adviser Approval

Associate Professor Joel Esposito
Weapons and Systems Engineering Department

_____
(signature)

_____
(date)

Acceptance for the Trident Scholar Committee

Professor Maria J. Schroeder
Associate Director of Midshipman Research

_____
(signature)

_____
(date)

1 Abstract

Unmanned vehicles have established an important place in the modern battlefield. They play a key role in intelligence and surveillance while not putting human lives in harm's way. The United States Navy recognizes this and has created *The Navy Unmanned Surface Vessel (USV) Master Plan*, which lists autonomous launch and recovery (L&R) as a key challenge.

A necessary enabling technology is the ability to automatically identify L&R sites from sensor data. This project focuses on the identification of suitable docking sites from three dimensional Light Detection and Ranging (LiDAR) scans. A LiDAR sensor is a sensor that collects range images from a rotating array of vertically aligned lasers.

Our solution leverages open source C++ code from Point Cloud Library -- "a standalone, large scale, open project for 2D/3D image and point cloud processing." Given a LiDAR point cloud our identification algorithm proceeds as follows. First the Random Sample Consensus (RANSAC) algorithm is used to isolate horizontal planar surfaces that may belong to the dock. After removing planar dock points, Euclidean Cluster Recognition is used to isolate point clusters that are potential vertical pilings. Bayes' Theorem is used to compute the probability that each cluster matches the characteristics of piling. For each candidate piling, the origin of that cluster is compared to the location of the target dock's planar surface. The dock can be identified by the relation of the pilings location to the dock's planar surface. The final output of the algorithm will be a sub-set of points, isolated from the original cloud, that are hypothesized to correspond to the dock.

Keywords: Algorithm, Dock, Locations, Point Clouds, LiDAR, Identify

2 Acknowledgements

Special thanks to:

Office of the Secretary of Defense for providing funding

MSC Graphics Lab

WSE Technical Support Division

3 Table of Contents

Contents

4 Background Information

4.1 Motivation

The strategic vision for the Navy and the Department of Defense includes an increased reliance on unmanned systems. Unmanned systems have demonstrated the ability to reduce cost because they are cheaper to build and maintain than manned systems. They also have the capacity to maintain persistent awareness for Intelligence, Surveillance, and Reconnaissance (ISR) missions, providing information on long-term trends about a target that would otherwise require numerous manned vehicles. This allows the commander to focus manpower on immediate threats. The benefits of unmanned systems have been outlined in *The Navy Unmanned Surface Vessel (USV) Master Plan*. In this plan, the systems are described as manual (manned), semi-autonomous (unmanned but with user inputs), or autonomous (completely unmanned).

One of the engineering challenges presented by the *USV Master Plan* is the Launch and Recovery (L&R) of the semi-autonomous and autonomous surface vessels. Whether it is from a stationary dock or another surface ship at sea, the United States Navy and the Department of Defense want unmanned surface vessels to be able to launch and recover on their own. For an unmanned surface vessel to recover itself after a mission, it must possess the fundamental capability of "dock recognition."

4.2 Sensor Information

The sensor used in this project to identify suitable dock locations is a three-dimensional Light Detection and Ranging (LiDAR) sensor. This sensor is an array of lasers that spins in a 360° horizontal field of view and collects range measurements of this surrounding area, see Figure 1 for an illustration of a LiDAR system. The sensor receives reflected energy from the lasers in order to calculate the range measurements. From these range measurements of the surrounding area, the data is then converted into Cartesian coordinates to be viewed as a three dimensional scene, referred to as a point cloud. Three-dimensional (3-D) LiDARs have proved themselves very useful on many autonomous ground vehicles, such as the Google Driverless Car Project, the DARPA, Defense Advanced Research Projects Agency, Grand Challenge, and the DARPA Urban Challenge (Bohren et al. 2008; Leonard et al. 2009). One last deciding factor on why the three dimensional LiDAR was chosen was for the fact that most autonomous vehicles are already using the sensor The algorithm will be taking data from a sensor that will already be on the autonomous surface vessel.

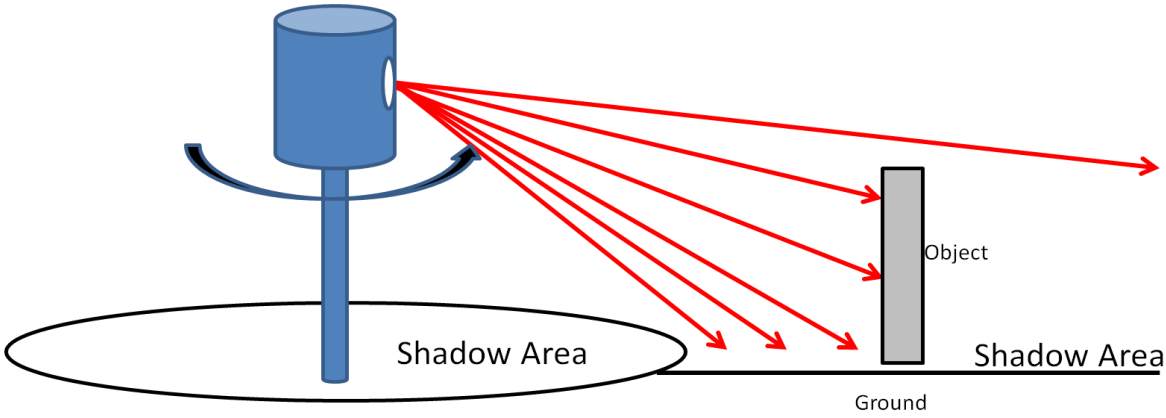

Figure 1: Illustration representing a LiDAR Sensor

LiDARs can be compared to Radio Detection and Ranging (RADAR) systems in the sense that both propagate energy and measure range to detect objects. The difference is that the propagated energy from LiDARs is from lasers while RADARs utilize radio waves; wavelengths propagating from lasers are short compared to radio waves. Short wavelengths imply a few differences; namely: (1) that LiDARs will have shorter range and higher scattering than RADAR; and (2) that very narrow beams can be formed, and with these narrow beams, comes higher resolution. The return that the LiDAR generates is a high-resolution reflection of the surrounding area. In Figure 2, the colors reference which lasers each range measurement is measured from. The red is measured by the lasers that are lower in the laser array, while the blue is measured from the lasers on the top of the array. This reflection can be used in mobile autonomous robots by programs that interpret the raw data provided by the LiDAR and then controlthe robot accordingly.

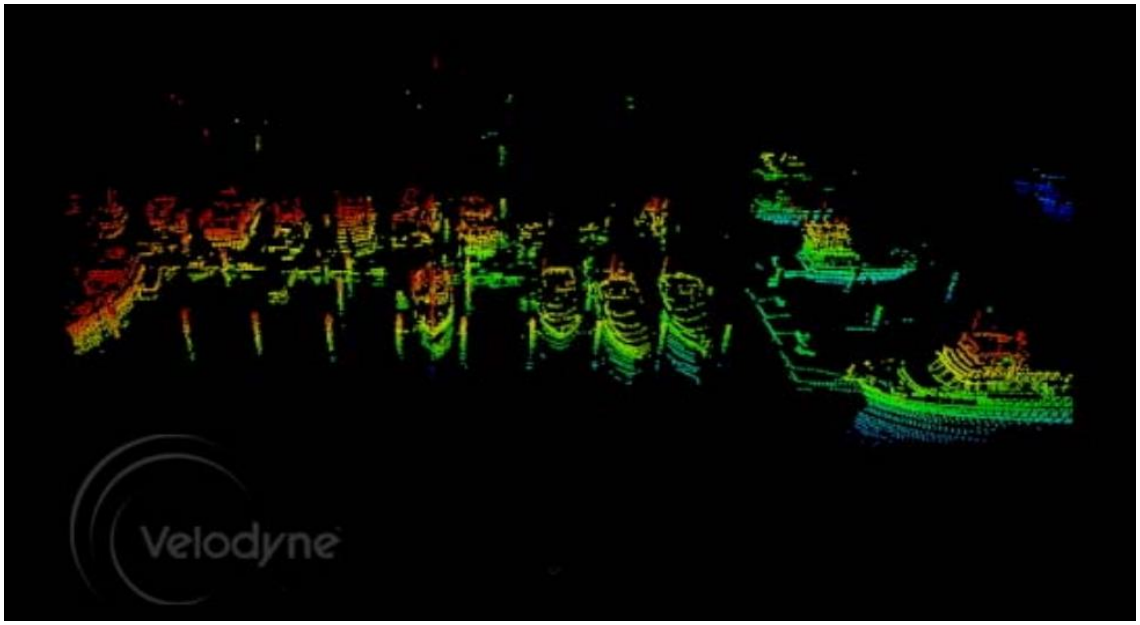

Figure 2: A LiDAR Scan of a pier with boats
(Maritime 2009)

Three dimensional LiDARs have the capability to measure ranges at high resolution. These two abilities combined, with the fact that using a 3-D LiDAR will not require any alterations to a dock, set it apart from any other possible sensors for this project. A project objective was to not modify the target docking locations because if the algorithm was dependent on a modified dock, the number of possible suitable docking locations would decrease. Figure 3 compares the 3-D LiDAR, to other sensors that could possibly have been used for this project; notice how the 3-D LiDAR matches the desired properties.

Sensors	High Resoution	Detect Range	Does Not Require Dock Modification
3-D LiDAR	Yes	Yes	Yes
RADAR	No	Yes	Yes
Cameras	Yes	No	No
Ultrasonic	No	Yes	Yes
Beacons/Transponders	No	Yes	No
2-D LiDAR	No	Yes	Yes

Figure 3: A table listing the possible sensors to use to identify a dock

The LiDAR that will be used for this project will be the Velodyne High Definition LiDAR - 32E, or the HDL-32E (Figure 4). The HDL-32E uses 32 lasers aligned from +10.67° to negative 30.67°in a vertical field of view, FOV. The HDL-32E uses a rotating head that gives it a 360° FOV horizontally. This allows the HDL- 32E to collect up 700,000 range measurements points a second that represents the three dimensional surrounding environment. The HDL-32E's lasers are emitted at a wavelength of 905 nanometers; the laser is Class 1 (Velodyne 2011). It provides range measurements up to one hundred meters with an error of +/- two centimeters while rotating

Figure 4: Velodyne HDL-32E LiDAR
(Velodyne 2011)

at ten Hertz. The Velodyne LiDAR also has an inertial measurement unit (IMU) containing accelerometers and gyros, by which it can estimate its own orientation. This can be used to correct scans taken on a platform that experiences pitch, roll, yaw, and heave.

The HDL-32E, and other LiDARs, share some limitations. Since the sensor is a range finder, it can only detect the first object that it hits. Everything behind that initial data point is unknown. This creates a shadow area behind the close objects. Another shadow area is created from the sensor's vertical field of view. The sensor cannot hit the objects beneath it because the bottom laser in the array does not aim straight down. This produces a circular shadow area directly beneath the LiDAR; refer back to Figure 1 for an image depicting the shadow areas. Also, the laser returns from very reflective objects are often inaccurate and can cause discrepancies in the raw data. In addition, transparent objects often produce no return. This can be seen back in Figure 2 where the water is not detected around the pier. This is because the transparency of the water diffracts the laser beams causing no return instead of reflecting the energy back towards the sensor like a normal solid object (Maritime 2009). This means that when the HDL-32E is placed on a boat, the water should produce no return. This would not be a problem since water is not considered an object; however, the LiDAR sensor picks up white caps and surface debris.

5 Experimental Details

5.1 Point Cloud Library (PCL)

The decision was made to use as many pre-existing algorithms as possible in order to save time and effort on this project. At the beginning of the academic year, we made a decision to use Point Cloud Library, or PCL. Point Cloud Library is a standalone, large scale, open project for 2D/3D image and point cloud processing. PCL provides a library of pre-written C++ functions for manipulating point cloud data, or PCD. Point cloud data files (.pcd) are used to store the data in this project; just as a picture file may be stored as a ".jpeg". Besides interacting well with PCD files, PCL offered many other capabilities that were deemed important in this project. Some examples are filtering data, registering data sets, and identifying key points. In the original project outline, the planar surface of the dock was to be identified using robust estimation algorithms, like Random Sample Consensus (RANSAC). The fact that PCL had a library that already possessed these capabilities made it an advantageous tool for use. Another benefit that comes from using an open-source library like PCL is that others may freely use the code in order update this algorithm, or use this algorithm to inspire their own ideas.

5.2 OpenCV

The final algorithm also uses image processing techniques to isolate the features of the desired docking locations. In order to gain the benefits of using computer vision techniques while not losing time by writing algorithms, it was decided that OpenCV would be used for the image processing steps just as PCL was used for the point cloud portions. OpenCV is an open source computer vision library written in C and C++ and works on many platforms and many specific coding languages (Bradski and Kaehler 2008). OpenCV provides simple-to-use algorithms for the image processing portion of this algorithm. The images that these image processing techniques will be used on are created in the algorithm from point cloud data and are from other sensors.

5.3 Data Collection

5.3.1 Target Dock

There are many types of dock designs. In order to make this project tractable it was determined that only one specific style of dock need to be identified. The targeted dock in the project has a raised planar surface surrounded by pilings. This is important because both level surfaces and pilings provide identifiable features in a LiDAR scan. The dock is identified using the pattern from the spacing between the identified pilings in relation to the planar surface of the dock. The reason a flat dock with no pilings is not a targeted docking site in this algorithm is because it lacks distinguishing geometric features. A parked barge or seawall could appear flat enough to register as a dock which is undesirable. The pilings are used to further differentiate the desired docking locations from other objects in the water.

For this specific project, it was established that most of the testing would be done off one dock. This target dock used is located on the northern side of Dewey Field behind Rickover Hall on the United States Naval Academy campus, see Figure 5.

Figure 5: Aerial view of the target dock (center)

5.3.2 Data Collecting Process

In order to collect data, the LiDAR would be placed on a cart and a MATLAB program was used to collect data of one scan, or one 360 degree rotation, of the LiDAR. These scans were saved as MATLAB figures for visual reference and as PCD files to be imported into PCL. The scans were collected from stationary positions. The decision was made to collect data using a cart instead of a boat because the cart allowed for the elimination of many variables, like the motion of the waves, along with the elimination of risk to the equipment. Along with the PCD files, pictures were taken of the surrounding area for use as reference. Lastly the scan location was referenced on an overhead shot of the area; please reference Figure 6 to see the location of each scan and Figure 7 for an example of the point cloud data collected and corresponding photos.

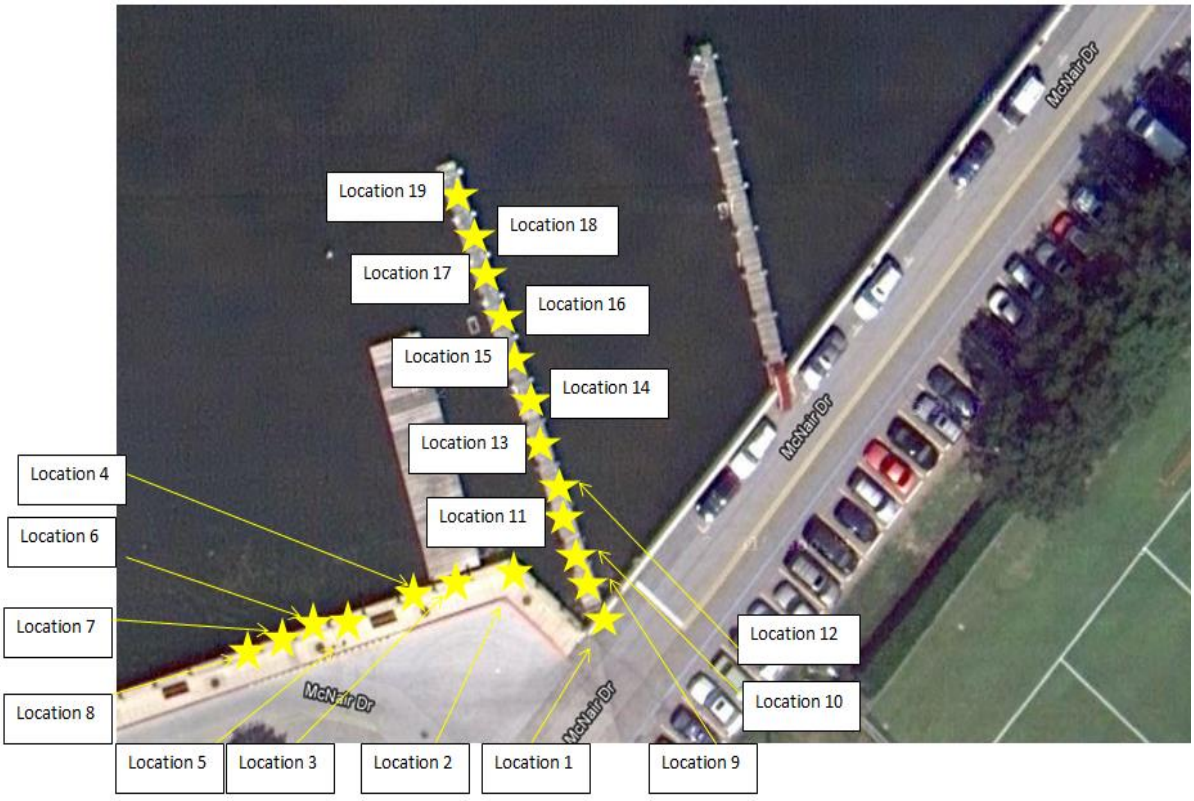

Figure 6: Locations of nineteen data scans
Imagery ©2013 U.S. Geological Survey. Map data ©2013 Google

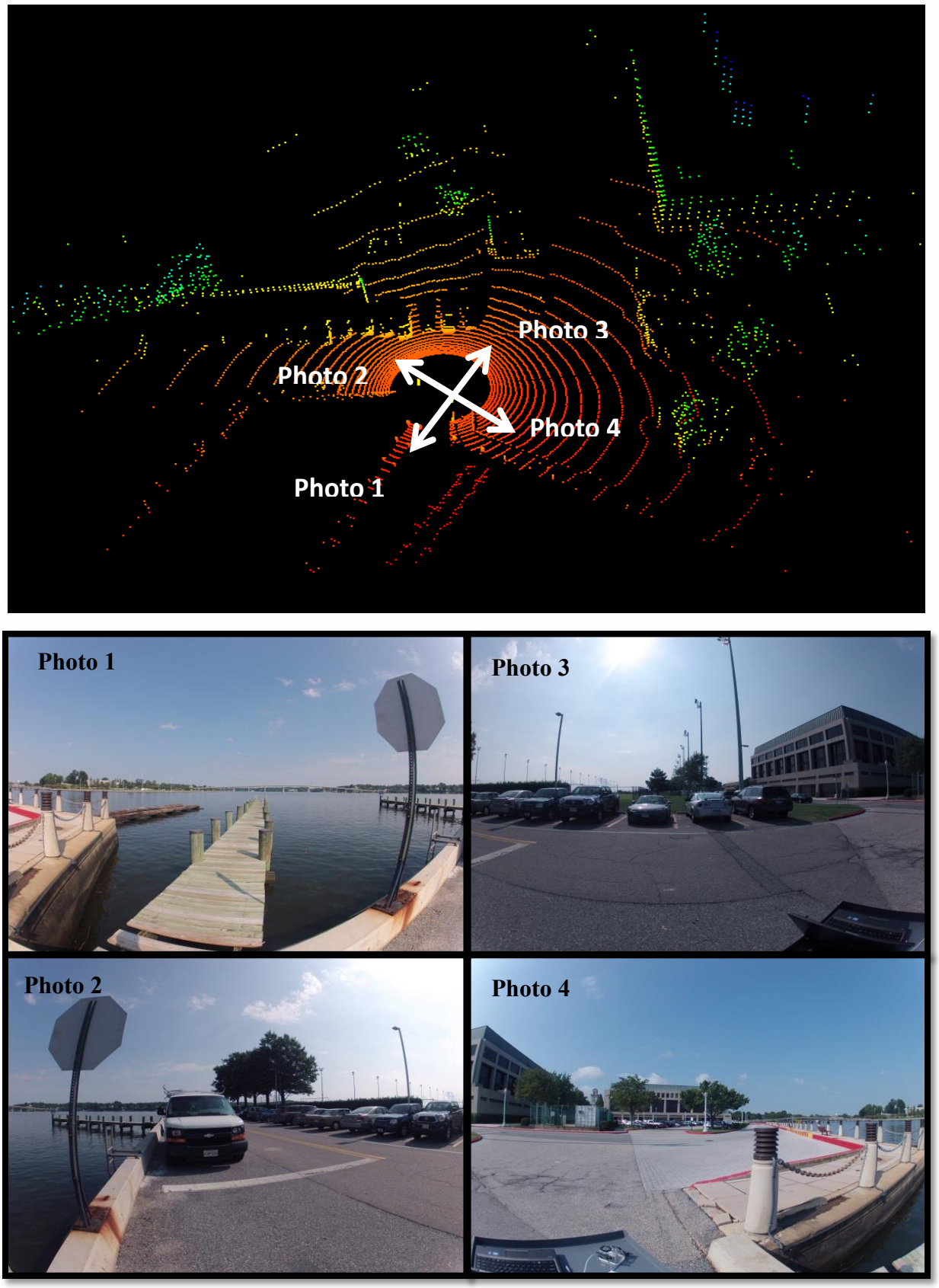

Figure 7: Top image is a LiDAR scan of position one from Figure 6 and the bottom half is four photos that correspond with the LiDAR scan

Initially when data was collected, the LiDAR was about five feet off the ground. The point clouds that were collected at this height were deemed unusable because the majority of the points collected corresponded to the sidewalk within two meters from the sensor. Very few points corresponded to the dock or other features commonly seen in a maritime environment. This was due to the LiDAR's field of view, or FOV. As stated earlier, the LiDAR's vertical FOV is mostly negative, meaning that most of the lasers are pointed down. The decision was made to raise the LiDAR off the ground because this would result in a dataset where the data points were located farther from the sensor. Figure 8 shows data collected at the different heights from the same location. It can be seen that the dataset collected at the higher altitude produces a larger scan. It also more accurately mimics the height in which the LiDAR would sit on an autonomous surface vessel if it were to be used in this application; typically, the height of a mast or where a RADAR is placed on a surface vessel is nearly twenty feet above sea level. The new height that the LiDAR was set to was about 15 feet. Figure 9 shows the cart with the sensor on a fifteen foot mast, while Figure 10 shows the cart on a dock in the process of data collection. Scans were then collected with the raised LiDAR and this produced more usable results, meaning that more returns corresponding to features over five meters away were collected.

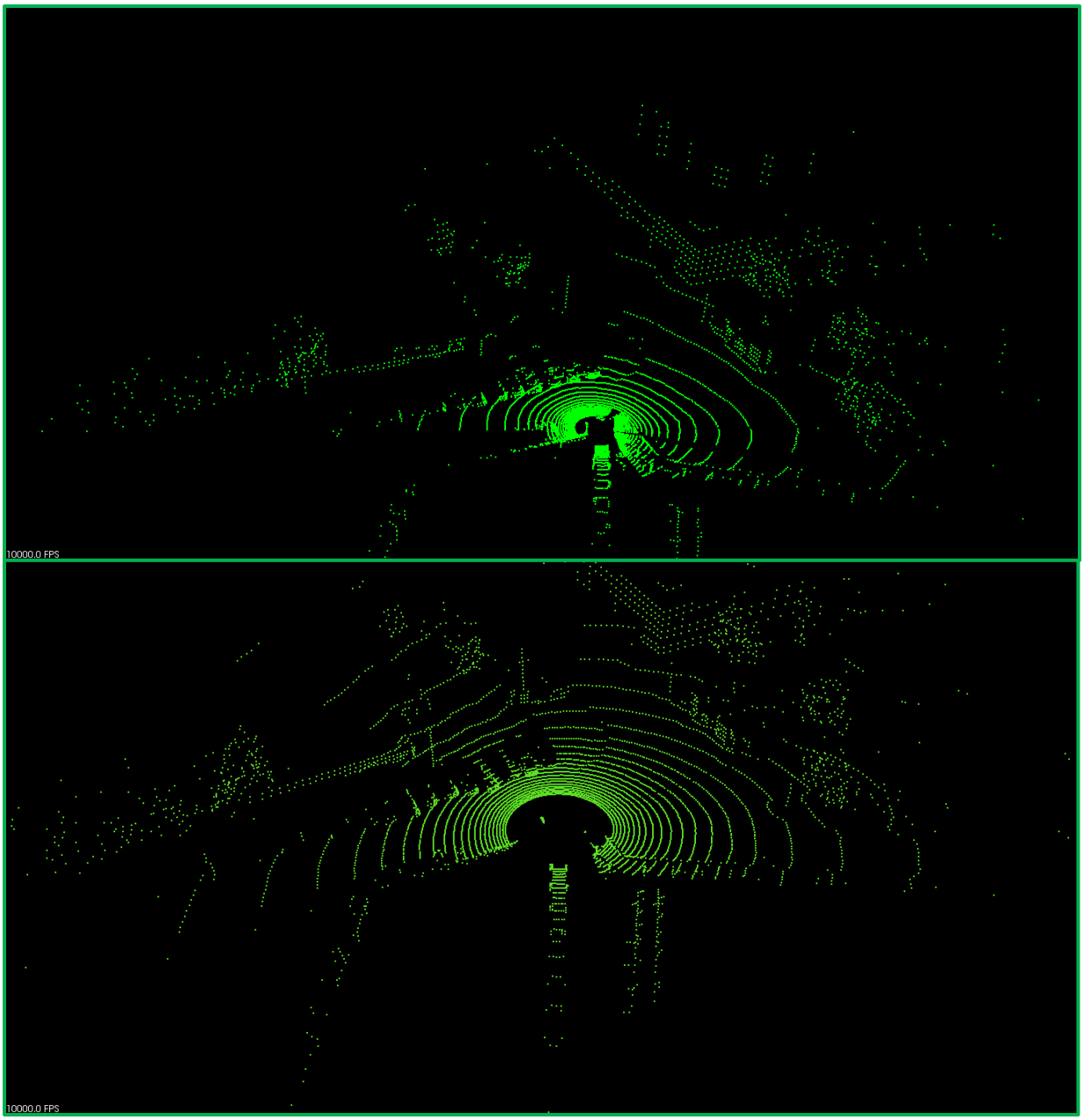

Figure 8: The top scan is from an elevated height of five feet, while the bottom scan is from an elevated height of 15 feet and contains more information about the surroundings

Figure 9: Data collection cart with the sensor elevated 15 feet.

Figure 10: Image of the cart during the process of collecting data.

These scans that include objects at longer ranges are more desirable; however, the additional scans from the new height produced a new problem. The resolution of the dataset was reduced in the dock region. This is due to the fact that the linear distance between adjacent laser readings increases as objects are farther away from the sensor. This can be seen back in Figure 8. Note that the center black circle is the shadow area mentioned previously. The distance between adjacent laser scans at twenty five meters away from the sensor is nearly four meters. This leaves gaps in the point clouds in which no data is collected. This is a problem because the proposed algorithm relies on identifying pilings, but pilings will not be identified if they fall into the data gaps caused by the divergence of the lasers. This leaves the task of somehow creating a higher resolution data set that will not miss any features in the dock.

6 Method

This algorithm is a multi-step process that utilizes both the 3-D features of the point clouds and the 2-D arrangement of the data points as seen from an overhead view. A quick overview of the method can be seen in Figure 11 below.

Dock Isolation Method Psuedocode

1. Acquire a Scan
2. Register the Scan
    a. SAC-IA Alignment
    b. ICP Alignment
3. Remove the Shore
4. Find the Planar Surface of the Dock
5. Identify Pilings
    a. Extract the Plane
    b. Use Probability to Identify Pilings
6. Compare Identified Pilings with Objects Removed from the Shore
7. Evaluate Identified Point Clouds

Figure 11: Dock isolation Method Psuedocode

6.1 Registration and Concatenation of Multiple Point Clouds

6.1.1 Registration

Raising the LiDAR sensor higher off the ground created scans that were larger in size since the lasers could then detect objects farther away. However, this compromised the resolution for each scan. Purchasing a higher resolution LiDAR was not an option. Also, taking scans from closer ranges so that the LiDAR would be able to identify more of the dock's features was undesirable because it eliminated the applicability this algorithm would have to autonomous surface vessels. Instead, the decision was made to register, or align, a series of scans, taken from different positions, into a single higher resolution point cloud. Figure 12 shows two different scans taken at two different positions three meters apart.

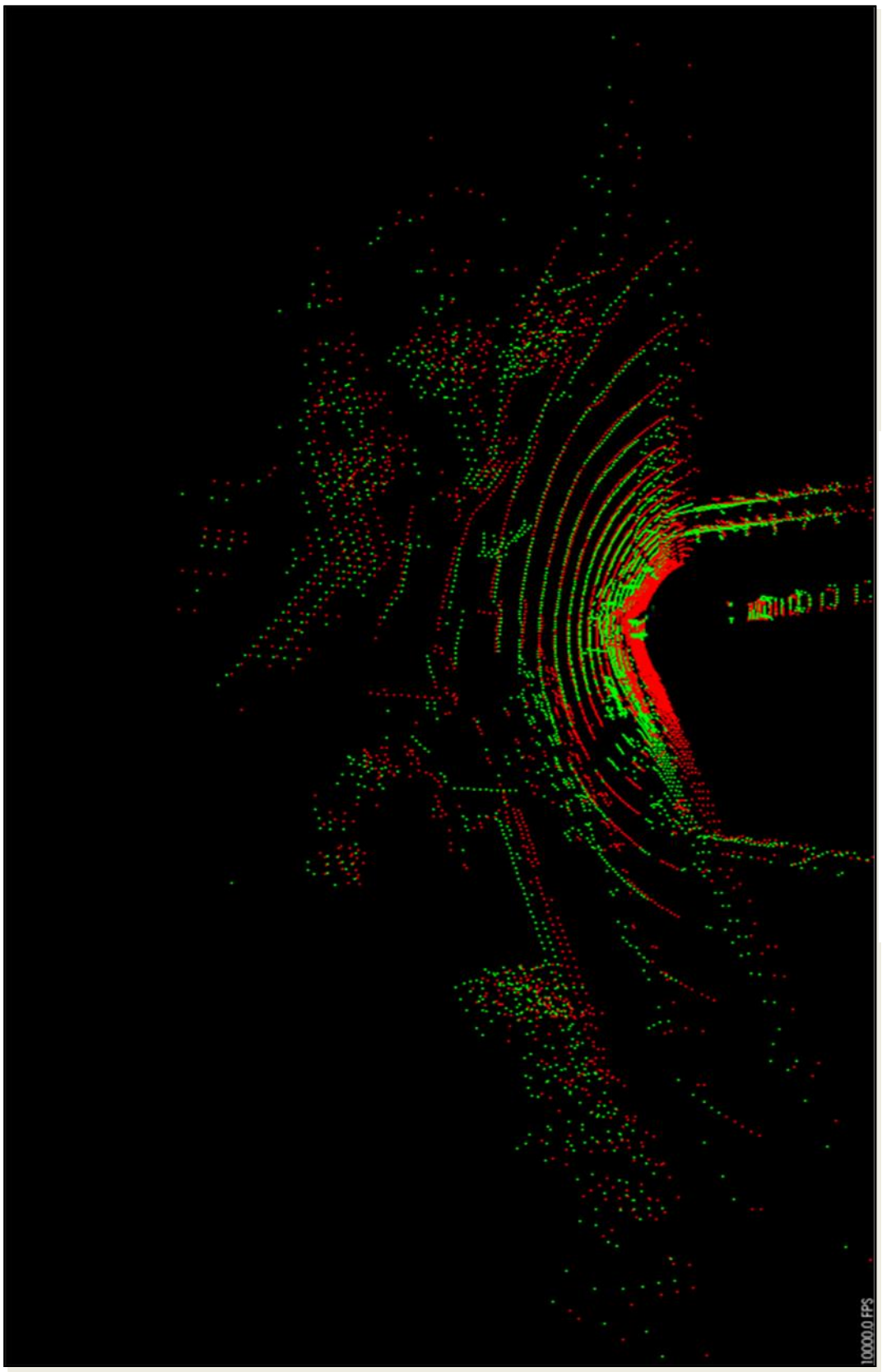

Figure 12: Two point clouds taken three meters apart.

Registration uses different mathematical optimization techniques to align the two different data clouds. The output of a standard registration function a homogeneous transformation matrix (M_i^{i+1}) that aligns point cloud i to point cloud i+1. A homogeneous transformation matrix is a mathematical representation of a rotation and translation between two reference frames, seen here:

$$M_i^{i+1} = \begin{bmatrix} R_{11} & R_{21} & R_{31} & T_{41} \\ R_{12} & R_{22} & R_{32} & T_{42} \\ R_{13} & R_{23} & R_{33} & T_{43} \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The top left three by three matrix represents one coordinate system's rotation in relation to another coordinate system. For example, $[R_{11} \quad R_{12} \quad R_{13}]^T$ represents the x axis of frame i expressed in terms of unit vectors (i.e. x, y, and z axes) of frame i+1. The three by one vector, T in Equation 1, represents the translation of frame, between the origin of frame i and the origin of frame i+1, expressed in terms of unit vectors. The bottom row is always three zeros and a one. The transformation matrix contains all the information that is needed to transform, or align, a point cloud to another point cloud. The other benefit of using transformation matrices is that multiple coordinate systems can be aligned through matrix multiplication. A quick example being if there are two transformation matrices, and one relating coordinate system "i" to coordinate system "i+1", and the other relating coordinate system "i+1" to coordinate system "i+2". One can quickly find the transformation from "i" to "i+2" by multiplying the two transformation matrices together;

$$M_i^{i+2} = (M_{i+1}^{i+2})(M_i^{i+1}).$$

PCL provides a library of registration functions that offers multiple ways to try to register different point clouds (i.e. estimate M_i^{i+1}).

6.1.1.1 Sample Consensus- Initial Alignment

Sample Consensus-Initial Alignment, or SAC-IA, is an algorithm that uses a random sample of points from the source point cloud and compares them to similar points on the target point cloud and the output is a homogeneous transformation matrix that would roughly align the two point clouds. The goal of implementing SAC-IA as part of the overall algorithm is to provide an estimated transformation that will be used as an initial guess for the Iterative Closest Point, or ICP, registration method (discussed in the next section) to find the global minimum of each registration. The way that this algorithm works is that there is a user defined number of random points chosen from the source point cloud. These random points have to a be user specified minimum distance from each other. For each one of these points, the neighboring data points, which are determined by a user defined radius, are used to estimate surface normals and features needed to help calculate the Fast Point Feature Histograms (FPFH) (Rasu et al. 1992). Using FPFHs, SAC-IA iterates through each random sample from the source point cloud and compares that point's features to points in the target point cloud until there are similarities. When this happens, for each random sample point in the source cloud, a list of points in the target cloud is generated based off similar histograms. Each point is randomly paired up with one of the similar points from the target cloud, which becomes its correspondence. A rigid

transformation is then computed between the sample points and their correspondences. An error metric is then used to compute the quality of the transformation. These three steps, seen in Figure 13, are repeated, and the transformation attempt that had the smallest error is stored as the output.

1. Selects user defined sample points (k) from source cloud while making sure the distances between points are greater than the user given parameter, d_{min}.
2. For each sample point, find a list of points in the target cloud whose histograms are similar to the sample points' histogram. From these, select one which will be considered that sample points correspondence.
3. Compute a rigid transformation defined by the sample points and their correspondences. Calculate an error that determines the quality of the transformation.

The error metric for the third step is determined using a Huber penalty measure: $L_h(e_i) = \begin{cases} \frac{1}{2}e_i^2 & \|e_i\| \leq t_e \\ \frac{1}{2}t_e(2\|e_i\| - t_e) & \|e_i\| > t_e \end{cases}$

Figure 13: SAC-IA Algorithm Psuedocode

6.1.1.2 Iterative Closest Point

The way that ICP registration method works is, given two partially overlapping input point clouds, one named the source point cloud and one the target point cloud, it uses local optimization to compute M_i^{i+1} in order to minimize the "alignment error". The algorithm also requires an initial estimation of transformation (which in this algorithm is provided by SAC-IA) and criteria to stop the iterations. ICP is an algorithm that iteratively adjusts the transformation matrix to reduce the mean square error between the target and source point clouds. A smaller average squared error would indicate a better fit. The algorithm has four steps, which can be seen in Figure 14 (Besl and McKay 1992).

The ICP algorithm has three criteria for termination: the number of iterations has reached the maximum user imposed number of iterations, the difference between the previous transformation and the current estimated transformation is smaller than a user defined value, or the mean of the Euclidean squared errors is smaller than a user defined threshold. ICP is extremely efficient; however, the problem with the Iterative Closest Point algorithm is that if the last two termination criteria are never met, the algorithm often settles into a local minimum instead of the desired global minimum. Essentially, the algorithm iterates back and forth around a local minimum and its surrounding points until the maximum iterations are reached. If the initial estimate of the transform matrix is poor, the ICP algorithm will never find the desired global minimum. Instead of inserting a random or generic guess for each registration attempt, it is more desirable to have an automatic initial approximate alignment transformation step. This transformation would have to be accurate enough that if ICP is then used, ICP can find the global minimum. This is the reason why an initial estimate is required as an input, and in this case is provided by the SAC-IA algorithm.

1. Get input point clouds. Declare one as the source point cloud and one as the target
2. While Termination Criteria == False
    a. For each point in the target cloud
        {
        Find the nearest point in the source cloud
        }
    a. Compute the transformation from the source point cloud onto the target point cloud
    b. Transform source point cloud
    c. Compute mean squared error

Figure 14: ICP Algorithm Psuedocode

Figure 15 below displays all the user defined values for both SAC-IA and ICP algorithms. Appendix A provides sample values for each user defined value on various clouds, along with the resulting transformation matrix. Figure 16 below shows the same two clouds from Figure 12; however, in this image, the source cloud has been multiplied by the transformation matrix that has been produced through the combined efforts of the SAC-IA and ICP registration methods resulting in two aligned point clouds.

User Determined SAC-IA Parameters

- Number of Samples: Sets the number of random samples (k) that will be selected from the source point cloud
- Normal Radius: Sets the radius for the normal search around each random sample.
- Feature Radius: Sets the radius for the feature search around each random sample.
- Minimum Sample Distance: Sets the minimum distance threshold between the randomly selected points.
- Maximum Correspondence Distance: Allowable error for code to terminate before maximum number of iterations is reached.
- Maximum Iterations: Set the maximum number of iterations for the algorithm to go through before termination.

User Determined ICP Parameters

- Maximum Correspondence Distance: Sets the maximum distance threshold between two correspondent points in the source and target clouds. If the distance between two corresponding points is larger than this threshold, the points will be ignored in the alignment process.
- Transformation Epsilon: The transformation epsilon is the allowable difference between two consecutive transformations. The algorithm terminates when the transformation epsilon falls below this user determined value.
- Maximum Iterations: Set the maximum number of iterations for the algorithm to go through before termination.

NOTE: Maximum correspondence distance for SAC-IA is not the same as ICP

Figure 15: User defined values for the SAC-IA and ICP Algorithms

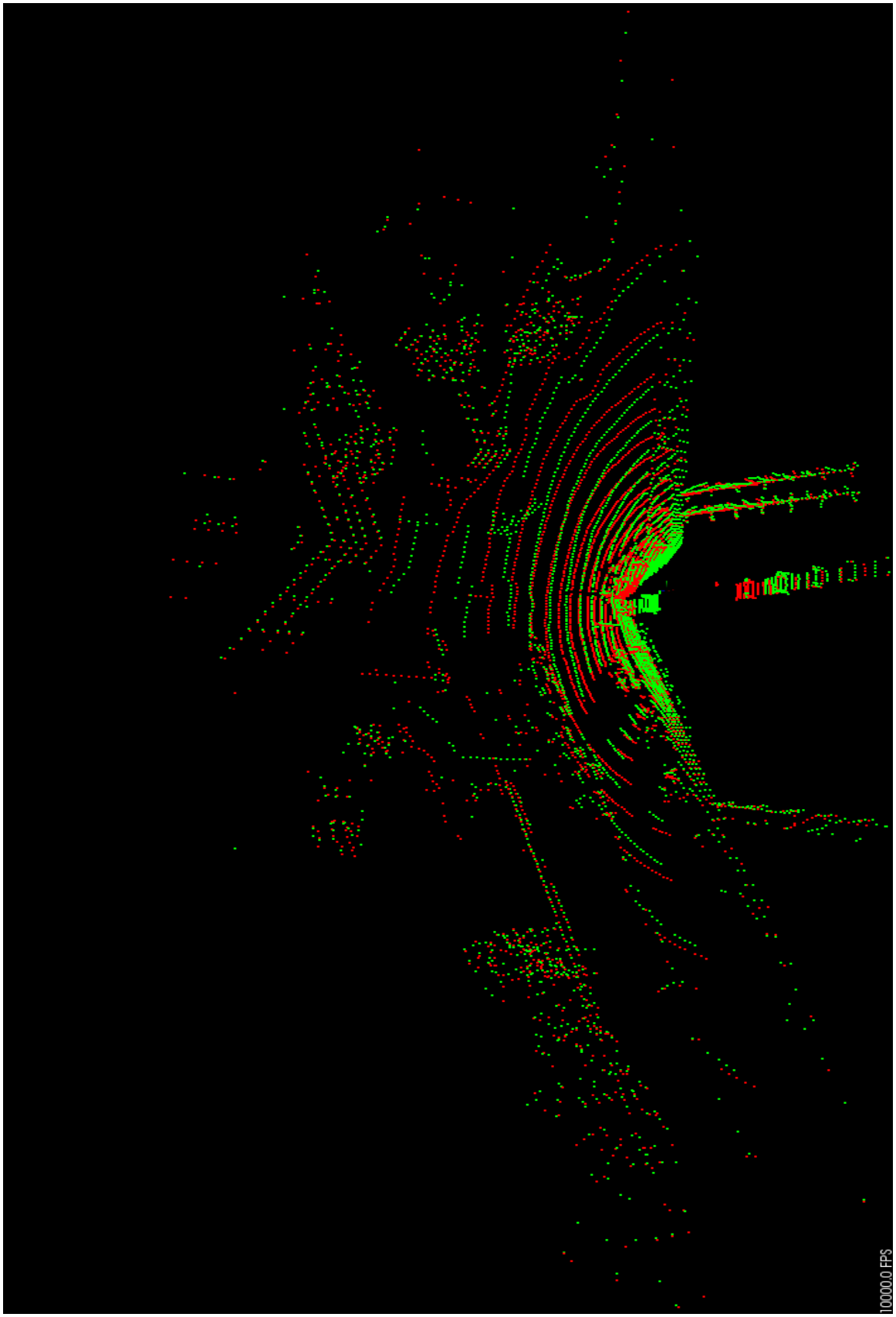

Figure 16: Two aligned point clouds as the result of SAC-IA and ICP registration algorithms

6.1.2 Concatenation

At this point, there are two scans that are aligned, but they are still two scans. The rest of the algorithm uses one dense point cloud that contains enough data to identify a dock as the primary input. Concatenation is the process of combining these two scans together. To facilitate the concatenation of scans, the process used was to select multiple point clouds that were collected in a line (such as down a dock or along the shoreline) and align them to one point cloud. This was done with six different point clouds spaced three meters along the target dock to form a point cloud in which a dock was easily identifiable. The resulting point cloud is robust enough to use to identify a dock. Figure 17 is a screenshot of six point clouds that were collected along the dock, registered, and concatenated together.

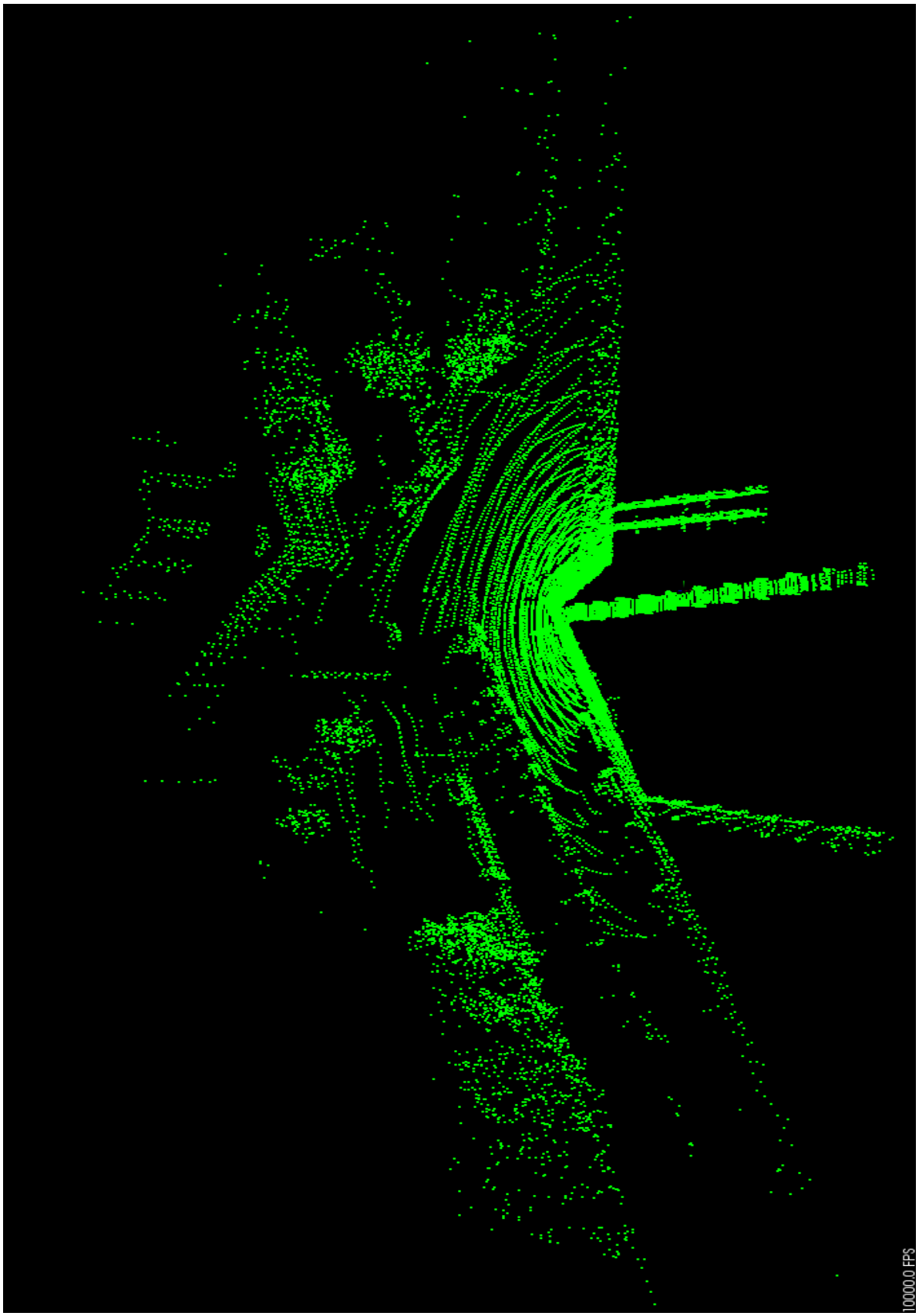

Figure 17: Dense point cloud created from six different scans

6.2 Shore Removal

Docks have distinct shapes in 3-D LiDAR scans. The LiDAR sensors are not able to obtain information from the water because the water refracts the lasers, so the data does not return to the sensor. The docks create thin peninsulas, as seen in the previous figure, which protrude from the shoreline. The objective of shore removal is to isolate these distinct shapes; if this can happen, it greatly reduces the amount of data that has to be examined by the rest of the algorithm. In order to remove the shore, image processing techniques were used.

Mathematical morphology is an image processing tool used for extracting image components that are useful in the representation and description of region shapes (Gonzalez et al. 2009). Many different logical operations fall under the category of morphology, two of which are used in this algorithm are dilation and erosion. These two processes are used on binary images. A binary image is a logical image in which the foreground is true (pixel value of 1) and the background is false (pixel value of 0). Dilation is an operation that "grows" or thickens" objects in a binary image while erosion is an operation that "shrinks" or "thins" objects in a binary image. The amount an object is "grown" or "thinned" is dependent on a user defined structuring element. A structuring element is a shape defined by a small matrix used to interact with the binary image that has a defined origin. For dilation, the structuring element's origin is translated throughout the plane of the image, and wherever the origin aligns with a 1-valued pixel, all the other pixels in the image that are under the structuring element also become a 1-valued. Erosion is almost the opposite, it iterates through all the pixels in the image, and when its origin lands on a 1-valued pixel the pixels inside the structuring element become zero valued.

Figure 18 shows eight images of each stage of this process, and the next section methodically goes over the steps in greater detail. In this algorithm, morphology plays a role in trying to remove the shore. The basic concept of what is done is that a binary image is created using the X and Y coordinates of every point in the cloud. Using only the X and Y coordinates of each point in the point cloud allows the algorithm to create a binary image of the overhead view of the area. This overhead view reveals the docks as penisulas and because of this unique feature, the algorithm tries to eliminate the shore through morphological processes. After the overhead image is created, the image is eroded using a structuring element that is twice as big as the target docks width plus one. The reason for this is to ensure that the entirety of the dock gets eroded. At this point, the eroded image is dilated in order to return the shore to its original size; however, this dilated image does not contain the docks, yet still retains the general shape of the shoreline. If this new image is subtracted from the original image, then the output image is the original image with the shore removed. This final image is then used to create a new point cloud based that is used in the next step of the algorithm.

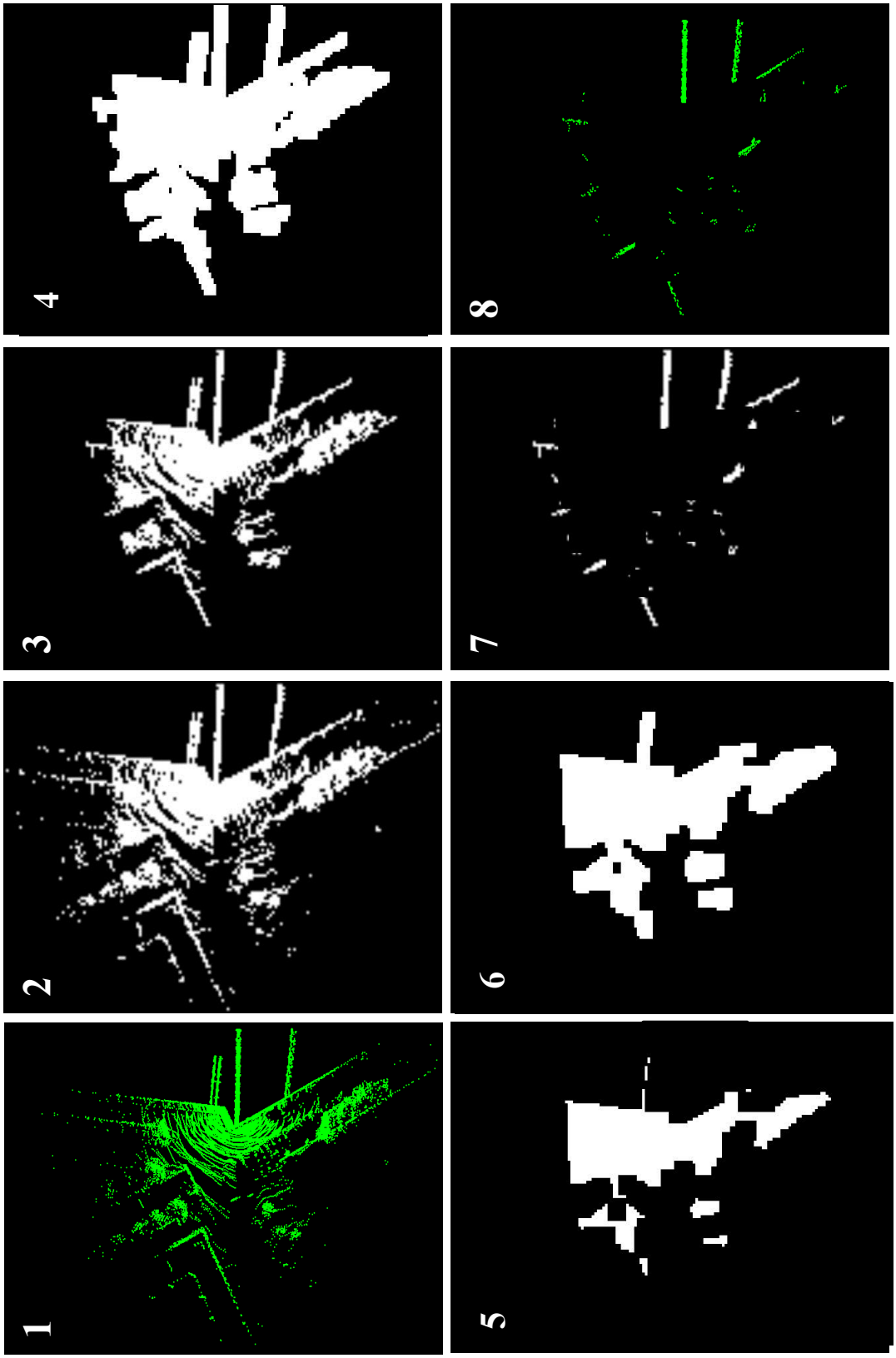

Figure 18: Eight images that correspond with the eight steps of shore removal

1. Image 1 in Figure 18 shows the original dense point cloud from an overhead view. This is to gain the perspective of the X and Y values of the point cloud that will be used to convert the point cloud into a binary image.
2. Image 2 exhibits the binary image obtained from the point cloud in Image 1 using its X and Y values. In this image each pixel represents a *1m x 1m* area of the point cloud, and if a point was determined to be in that one by one meter area, then the pixel becomes a 1, or white. Figure 19 shows the pseudocode for the conversion between the point cloud and the binary image, the C++ code can be seen in Appendix B.

Point Cloud to Binary Image Conversion Psuedocode

a) Determine minimum and maximum X and Y values of the point cloud
b) Round the maximum values up to the next integer and round the minimum values down to the next integer
c) Obtain the absolute size in the X and Y direction
d) Set pixel size to correspond to a real world distance
e) Create a binary image using the each values absolute size, but add a value of two in each direction so that no part of the point cloud lies on the edge of the binary image
f) For every pixel in the Y direction

```
        {
        For every Pixel in the X direction
                {
                For every point in the point cloud
                        {
                        If the X and Y coordinates of the point cloud
                        correspond the current pixel
                                {
                                Make the current pixel true
                                        }
                          }
        }
}
```

Figure 19: Point cloud to binary image conversion psuedocode

3. Image 3 displays the newly created binary image, but with the smaller individual pixels removed. Small groups of 1-valued pixels in a binary image can often cause undesired results when using morphological functions. This is because these small clusters of pixels often have little to no utility when trying to identify a dock; however, when morphological functions, such as dilation, are applied, they can connect to larger objects in the image. In the case of this algorithm, these individual pixels represent data points from the LiDAR scan that are not important to identifying the dock. It is assumed that at this point in the code, the dock and all of the information that needs to be extracted from it will be represented in the image by more than one pixel. This is why small, non-connected objects are removed at this point. In order to do this, the area of each object in the image is determined by using OpenCV contours. If the area of any given shape in the image is smaller than a user defined value, then those pixels are removed. In this algorithm, the defined value was twenty pixels.
4. The next step in the process is to dilate the image in order to fill in the gaps in the objects so that the whole foreground is one object. If there are gaps in the middle of the objects, it was determined that the gaps were part of the shore anyway, so cleaning

the image up and filling those gaps creates a cleaner output at the end of the process. In MATLAB there is a function called "imfill" which does just that; however, when working with C++ and OpenCV, there is no such function. Instead of finding pre-written function or writing a new one, it was determined that a simple dilation could complete the objective while not interfering with the overall process. The structuring element that was used represents a 2.25 by 2.25meter square. This means dilation process is expanding the shore by 1.125 meters in every direction and filling in the middle holes.

5. The process from Image 4 to Image 5 is two iterations of erosion using the same size element. The dock is no more than two meters across, so by using a structuring element that reduces the image in size by a relative 1.125 meters, the dock will be eliminated on the second iteration. The first iteration takes the previously dilated image back to its original size; however, the second erosion is what eliminates the docks that are thinner than the structuring element.
6. Image 6 is the eroded shore, with the two skinny docks removed, dilated one iteration by the previously sized structuring element. This dilation swelled the eroded shore back to the approximate size of the shore in the original image.
7. Image 7 is Image 6 subtracted from Image 2. This step essentially removes the shore from the image leaving the thin docks as the two largest objects. There is a lot of other smaller objects that are part of Image 7, but that is a reasonable expectation. It is expected that no morphological process will be robust and perfect when applied to varius scenes.
8. Image 8 is the overhead view of the point cloud that is created from the binary pixels in Image 7. For this step, it iterates through every point in the original point cloud and includes all points whose X and y values correspond to the true pixels in Image 7. This cloud now has fewer data points and will be used as the input for the next step of the algorithm, which is finding the plane of the dock.

6.3 Finding the Plane of the Dock

In the assumptions, it was stated that each dock would have a planar surface surrounded by repeated pilings. Identifying the plane and removing it allows the pilings to be identified as explained in the next section.

In order to find the planar surface in the point cloud, Random Sample Consensus (RANSAC), is the algorithm used to estimate the best fit plane to the data set. RANSAC is an iterative method that estimates parameters of a mathematical model (i.e. the equation of a plane in this case) given a set of noisy input data that contains both inlier and outlier data points (Fishler and Bolles 1981). Point Cloud Library has a class named "Segmentation". The class's job is to be able to segment desired data from a larger input data cloud. Given an input data set, the goal of the segmentation class is to find inlier data points that meet certain criteria. The criteria are based on the Euclidean distance of the points from the best fit plane model. The way the PCL implementation of RANSAC works is that the user has to input a model type, a method type, a threshold, and maximum iterations. The model type is the object that one is trying to segment from the point cloud, as stated earlier, this is the model of the plane. Method type is which robust estimator to use; in this case, RANSAC is being used. The threshold is the distance

allowance to find inliers. For this algorithm, the goal is to find the best fit plane using a 0.25 meter threshold. This was determined as the best threshold because it captures most of the planar surface of the dock, but still left the pilings as outliers. Figure 20 below shows the input cloud after the shore removal process, while Figure 21 shows the same cloud below, but with the planar inliers highlighted as red.

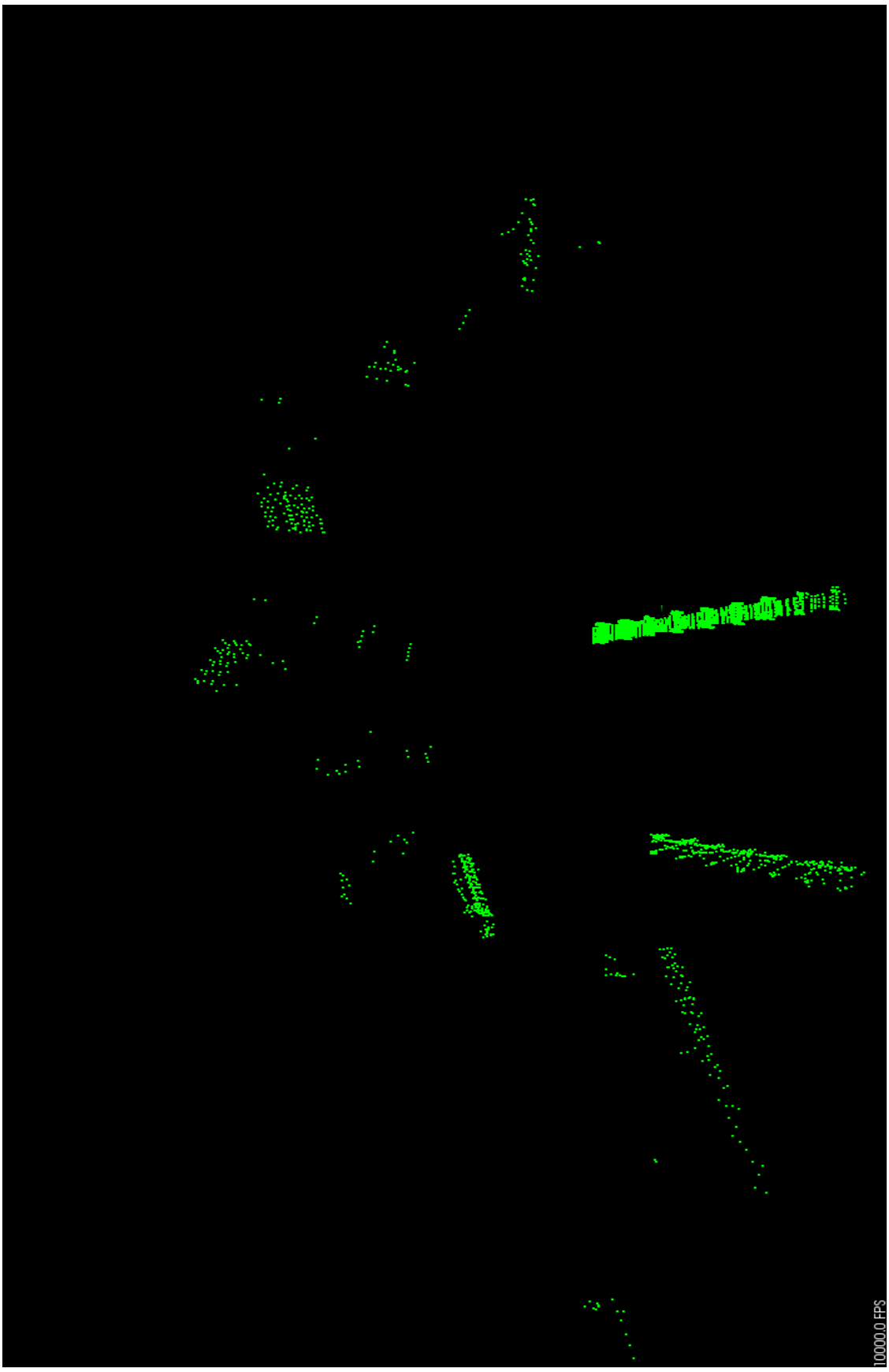

Figure 20: Point cloud with eliminated shore

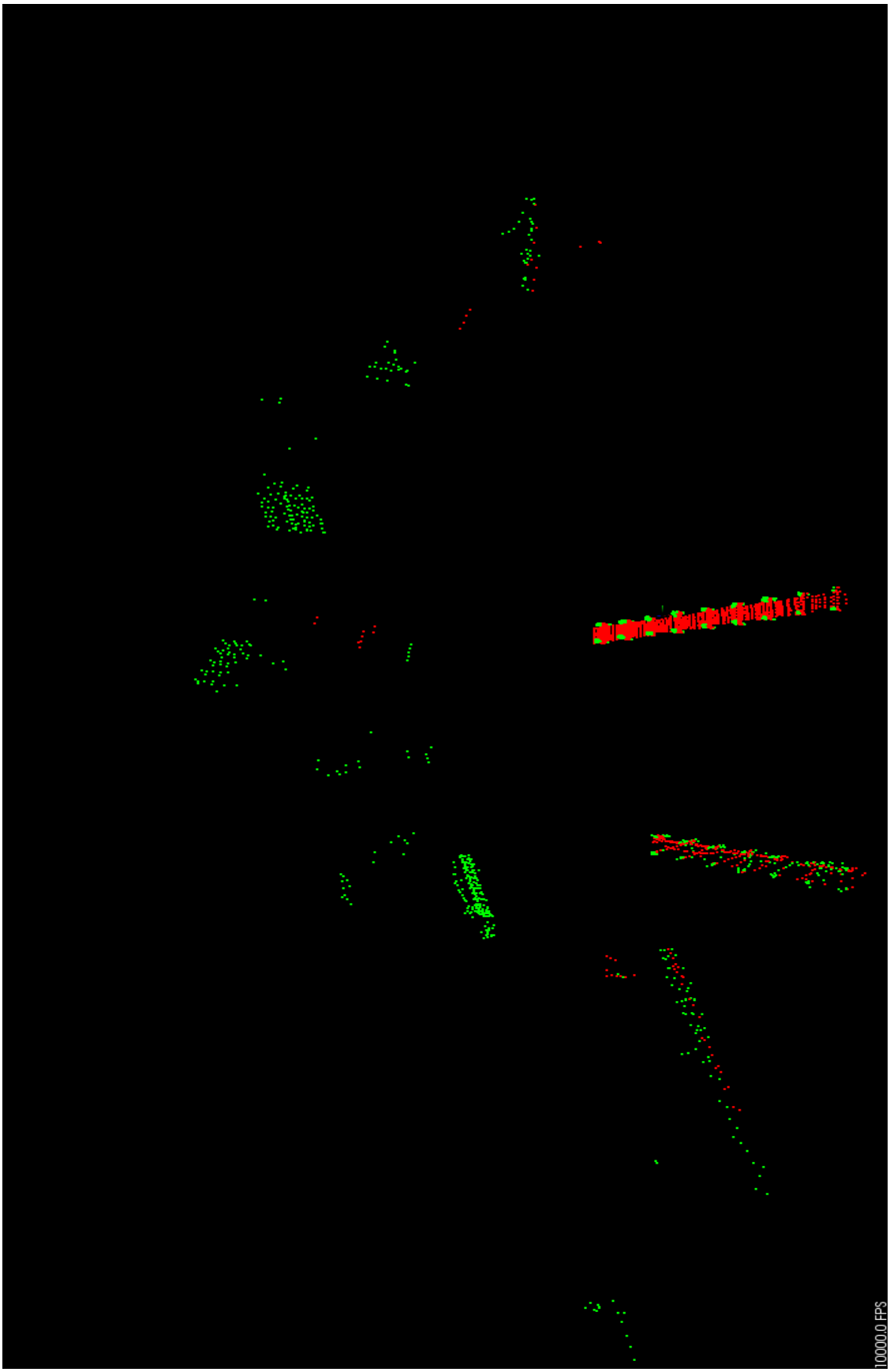

Figure 21: Shows the point cloud from Figure 20 with the dock identified in red

6.4 Identifying the Pilings

6.4.1 Extracting the Plane and Cluster Recognition

The first step to identify the pilings is to remove the recently found plane. The reason for doing this is that once the plane is removed, the pilings will be the only data points remaining of the dock. The segmentation class of Point Cloud Library allows for the extraction of points from a point cloud, so to extract the plane, all that needed to be done was extract the point cloud that contained the planar inliers. The result can be seen in Figure 23.

At this point, the next step is to cluster the data points of each piling together. Point Cloud Library offers an Euclidean Cluster Extraction as part of their segmentation class. What Euclidean Cluster Extraction does is divides an unorganized point cloud into smaller organized parts. These organized clouds can then be individually evaluated to determine if they meet the parameters of specific objects, such as pilings Feng and Song 2008). These smaller organized parts are determined by user inputs. The goal of implementing Euclidean Cluster recognition is to seperate each piling as its own cluster and then evaluate its shape to determine if it is a piling. The Euclidean Cluster Recognition function that is used in this algorithm applies a neighbor search technique. This means that for each point in the input cloud (the point cloud with the plane removed, see Figure 23), the points are evaluated to find its neighbors within a distance threshold. The distance threshold represents the radius of a sphere that encloses the given data point. If a neighboring data point lies within this spherical area, the two points are grouped together into a cluster. This process continues from each point in the inputted cloud until every point is part of a cluster. For this algorithm, the distance threshold was 30 centimeters. This distance was chosen because as seen in Figure 22, when the plane is removed, it extracts the data from the piling with it. Thirty centimeters allows the data points to cluster together over the chasm caused by the extraction of the plane, but does not allow the data points of one piling to cluster with another piling since the distance between pilings is three meters. Next the cluster size is evaluated. If the cluster has fewer than five data points, or more than two hundred, the cluster is dropped. No identifiable piling will have fewer than five data points, and none of the pilings have more than two hundred points. This eliminates some excess data before it is further processed.

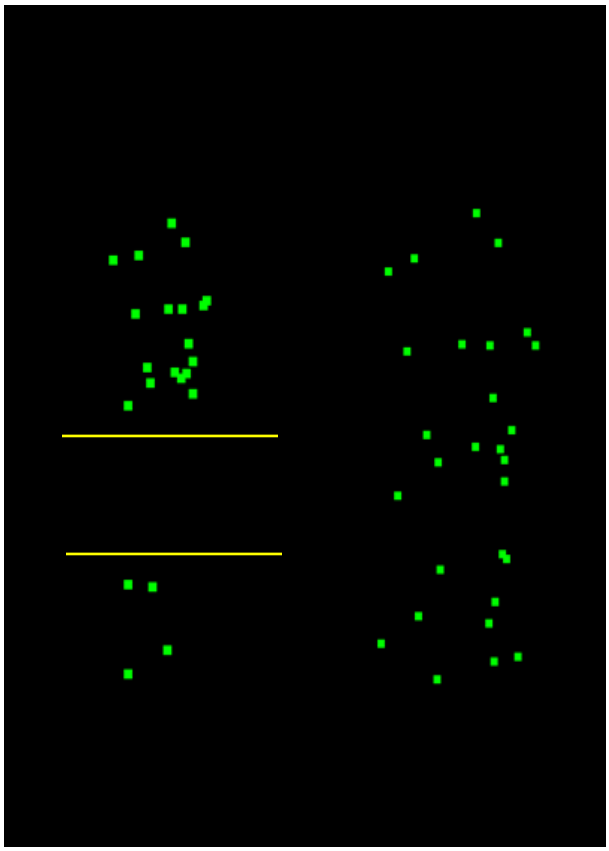

Figure 22: Piling on left has planar data points removed (area between the yellow lines) while piling on right has data points from plane

The next step is to obtain the whole piling. As previously mentioned, when the

planar inliers are extracted, it leaves a gap in the piling data, which means that the current cluster does not represent a whole piling. This step is to go back and add the data from the plane to the clusters. In order to do this, the code iterates through each cluster and determines a bounding box in the X and Y plane. Any point that is in the planar point cloud that is within the limits of a cluster's X and Y bounding box is then added to that respective cluster. Figure 22 shows the before and after effects of this part of the code on a piling close up. The second image clearly has more points and is more representative of a piling than the first image.

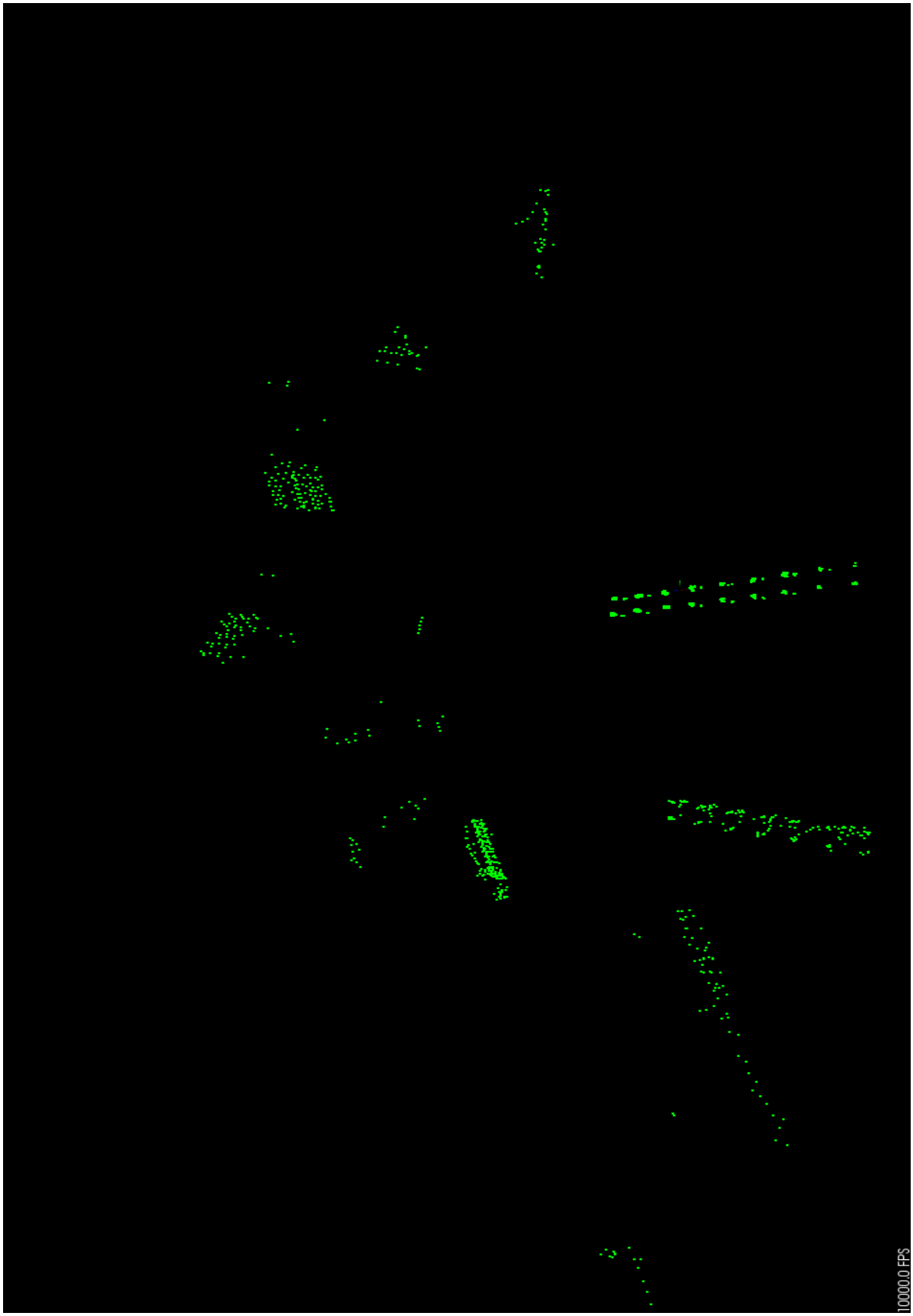

Figure 23: Point cloud from Figure 21 with plane removed

6.4.2 Using a Gaussian Probability Density Function and Bayes' Theorem to Identify Pilings

Once each cluster has had the planar data points re-inserted, the next step to identify the pilings is to evaluate each cluster to determine the probability it is a piling. Past research has shown that using Bayesian Pattern Recognition and other probability functions have proven successful when trying to identify different shapes in unorganized point clouds (Schnabel et al. 2006). Once each cluster has a probability of being a piling associated with it, only those that meet a user defined threshold are considered pilings.

The first step to identifying a piling is determining what features are useful to differentiate a piling from other clusters. Seven features were determined to be useful in this regard:

1. centroid height;
2. vertical separation of data points in the piling (height of pilings);
3. large diameter in the X-Y plane;
4. short diameter in the X-Y plane;
5. ratio of the two diameters;
6. ratio of the height of the piling over the ratio of the diameters; and
7. ratio of the height over the average of the diameters.

In order to compute these features, the centroids and three-dimensional covariance matrices were determined for each cluster. A three dimensional covariance matrix is 3x3 matrix. The diagonal of the covariance matrix represents the variance in each dimension, where the [1,1] element would represent the variance in the X direction while [3,3] element represents the variance in the Z direction. Below are the reasons why each feature was chosen and the steps to obtain that feature for each cluster.

1. Centroid Height – This feature was chosen because the pilings are always located near the water, so they will always show up around the same height. By evaluating centroid height as a feature, it takes vertical position into account and can help eliminate any clusters that may resemble a piling but not match the vertical position of a piling, such as a tree trunk. This value is obtained by simply taking the Z coordinate from the previously calculated centroid.
2. Vertical Range of Data Points in the Piling (Height of Pilings) – This feature is the approximate height of the piling. Evaluating this feature for each cluster identifies the vertical separation of the data points which is advantageous for eliminating clusters that are too tall or too short. To obtain this value, the information is extracted from the covariance matrix that is determined for each cluster. By obtaining the third row and third element of the covariance matrix produced by each cluster, the variance in the Z direction can be obtained. The square root of the variance is then taken to obtain the standard deviation. The standard deviation is then multiplied by six to determine the approximate height. Assuming a Gaussian distribution, three standard deviations is a plus or minus 98.5% confidence interval. This step utilizes the eigenvalues in the X-Y plane in order to find the diameter of the pilings. A perfect piling should have the same diameter throughout the piling because a piling is essentially a cylinder. However, what happens with point cloud data is that a LiDAR scan can normally only see half the piling on one scan. That is why features three and four are long and short diameters. In order to obtain the diameters of each cluster, the top left two by two matrix of the calculated three dimensional covariance matrix is extracted. This two by two matrix shows only the

variation between the X and Y dimensions. Using this covariance matrix, two eigenvalues are calculated. One represents the variance of the longest distance across the X-Y plane of each cluster while the other one represents the shortest value's variance. Using the same statistical technique applied in step two where the square root of the variance value is obtained then multiplied by six, the approximate values for each cluster's long and short diameter are determined.

3. The fourth feature calculated for each cluster is the shortest diameter. This is calculated the same way as feature 3.
4. The fifth feature is a ratio of the two diameters. The ratio is a good indicator of how symmetrical the cluster is. This ratio was calculated by simply diving the large diameter by the short diameter.
5. Feature six is the height of the data points from feature two, divided by the ratio of the diameters. The reason this is a feature, along with feature seven, is that in general, Bayesian classifiers will have greater discriminatory power when a larger number of features are used, especially when the features are not linearly dependent.
6. The seventh feature is the ratio of the height of the data points divided by the average between the two determined diameters.

The equation for the Gaussian Probability Density Function (PDF) is

$$p(\bar{x}|w_1) = \frac{1}{(2\pi)^{\pi/2}\left|C_j\right|^{1/2}} e^{-\frac{1}{2}(x-m_j)^T C_j^{-1}(x-m_j)}$$

where C_j and m_j are the covariance matrix and mean vector of the pattern population of class w_j and $|C_j|$ is the determinant of C_j. In general, a Gaussian PDF computes a PDF value evaluating one data set to a known or "trained" dataset. The output PDF value is high if the two datasets are similar, and likewise low if the two datasets are different. In this equation, class j represents the "trained" data. In order to implement Bayes' Theorem, each cluster needs to be evaluated through two Gaussian PDFs, one evaluates the probability that the cluster is a piling, and one calculates the probability that the cluster is not a piling. For each of these PDFs, supervised training data about each class must be collected. This means that data has to be gathered from clusters that are known pilings in a point cloud, and likewise for clusters that are known non-pilings. The Gaussian PDF equation is also useful for determining the global PDF value, p(x), which is a global PDF value for all classes combined. This value is later used in Bayes' Theorem.

The training data was collected by selecting certain clusters from the data that were known to be pilings or not pilings. For each known cluster of either class, the features listed above were calculated. Then a covariance matrix and mean vector were calculated from the data for each class. The covariance matrix and mean vector for each class were then used in their class's respective PDF for calculating the probability of that class. Using the known information for each population class, the PDF value of a cluster for each class is determined by evaluating that cluster's features in each of the PDFs.

Once the probability for each class is determined for one cluster, the algorithm utilizes Bayes' Theorem to get the overall probability that the cluster is a piling. Bayes' Theorem, produces a posterior probability that the observation $\bar{x}$ is a member of class w_1, seen below:

$$P(w_1|\bar{x}) = \frac{p(\bar{x}|w_1)*P(w_1)}{p(\bar{x})}.$$

In this algorithm, Bayes' Theorem will only deal with two population classes, "Pilings", w_1, and "Not Pilings", w_2. For Bayes' Theorem to work, each of these two classes requires a prior probability, or the likelihood of encountering a particular class within the overall population. This captures the frequency with which pillings appear in a typical point cloud data set, relative to other clusters such as cars, trees, boulders, etc. In this algorithm, these values were determined by examining an ideal point cloud data set and determining that there were twenty-two pilings that should be identified with the given information out of 125 clusters. From this information, the prior probability was set to be:

$$P(w_1) = (^{22}/_{125}),$$

$$P(w_2) = (^{103}/_{125}).$$

Once each cluster is evaluated and a probability of that cluster being a piling is produced, the algorithm selects all the clusters whose probability is over a certain threshold. For the clusters that are over the threshold, they are combined into a single point cloud while the other clusters are discarded. The threshold that is used in this algorithm is a 50% value, meaning that if Bayes' Theorem believes that there is greater than a fifty percent chance that the cluster is a piling, the algorithm then labels the cluster as an identified piling. Figure 24 shows the identified pilings from the current point cloud highlighted in red.

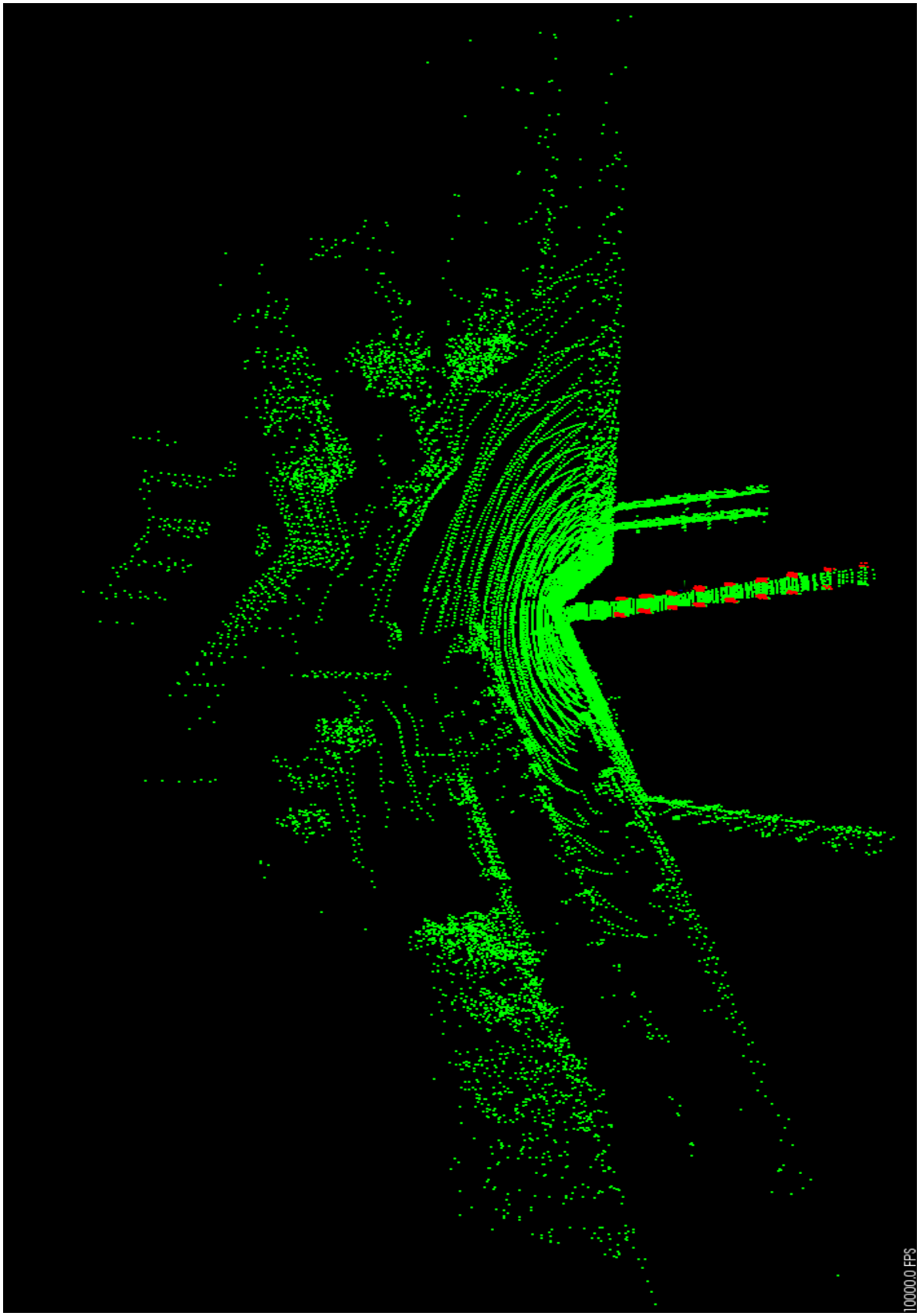

Figure 24: Identified pilings are highlighted in red

6.5 Compare Identified Pilings with Objects Removed from the Shore

The next step of the algorithm is to check the spatial arrangement of the pilings. One of the assumptions is that each dock identified will have pilings regularly spaced along each side of its planar surface. Morphology is used again to check the proximity of pilings. The algorithm assumes that the if the pilings are close enough to each other, then a dilation function will create a single object in a binary image. The area that is then identified with the pilings is multiplied with the binary image of the shore eliminated. This multiplication ensures that the only points that are converted into the final point cloud are points that are not part of the shore and are in between the closely spaced pilings. This process only has six steps, which can be seen in Figure 25. The steps are as follows:

1. Convert the point cloud of the pilings (Image 1) to a binary image (Image 2). This is using the same process that was used for the shore removal step.
2. The next step is to dilate the objects in the image. The goal is to dilate each piling enough so that it will connect with its neighboring pilings when they are dilated. Since the distance between pilings is three meters, the structuring element size represented 1.75 meters. This allows closely spaced pilings to become one object.
3. The next step is to erode the objects. Instead of eroding the objects using the same size structuring element as in the dilation step, the objects are eroded using a smaller sized structuring element. The purpose of this is to allow a buffer zone around the pilings for the next step. The structuring element used erodes pixels that represent 0.75 meters of the dilated object. This leaves a one meter buffer zone around the original pilings. Image 3 in Figure 25 shows the result of steps two and three.
4. The next step is to multiply this image of the combined pilings by the previously obtained image of the removed shore, Image 4. This allows only the pixels through which are both true, Image 5. This is the reason it is acceptable to have a buffer zone around the pilings from step three.
5. For all the pixels that remain true from step four, create a point cloud that represents all the points in the final binary image's corresponding region.

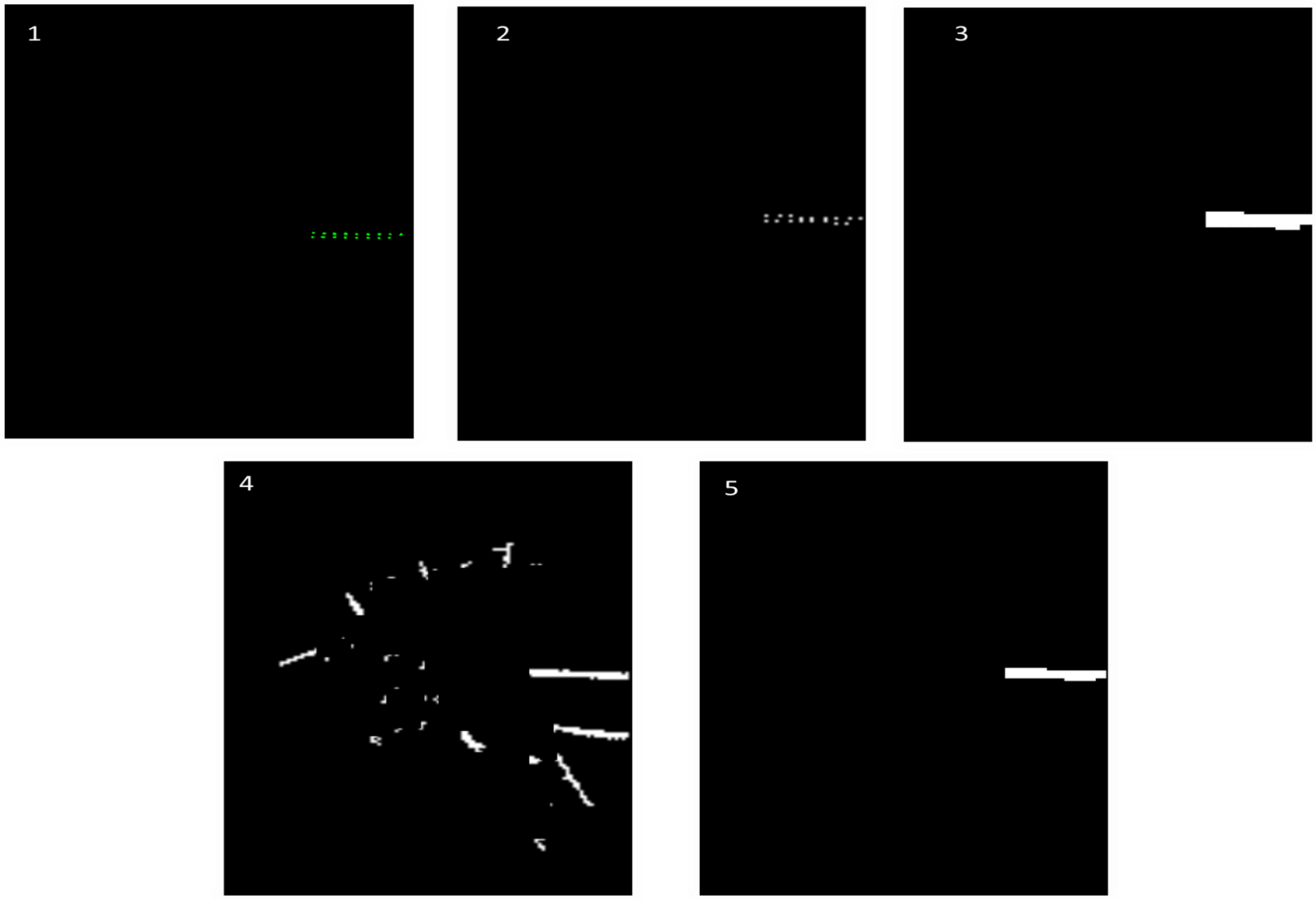

Figure 25: Morphological steps to check roximity of pilings.

6.6 Evaluate Identified Point Clouds

In some tests of different point clouds, it was noticed that sometimes the piling identification step would produce false positives, or it would find a real piling on another dock, but could not identify its neighboring pilings. This created issues with the final point cloud, see Figure 26. If there was a lone piling and that is not part of the shore, the algorithm would allow its surrounding points to be identified as a dock. The problem with this is that, while the piling may be part of the dock, it is not a suitable dock location, which is what the algorithm is truly trying to identify. This last step is to eliminate these issues.

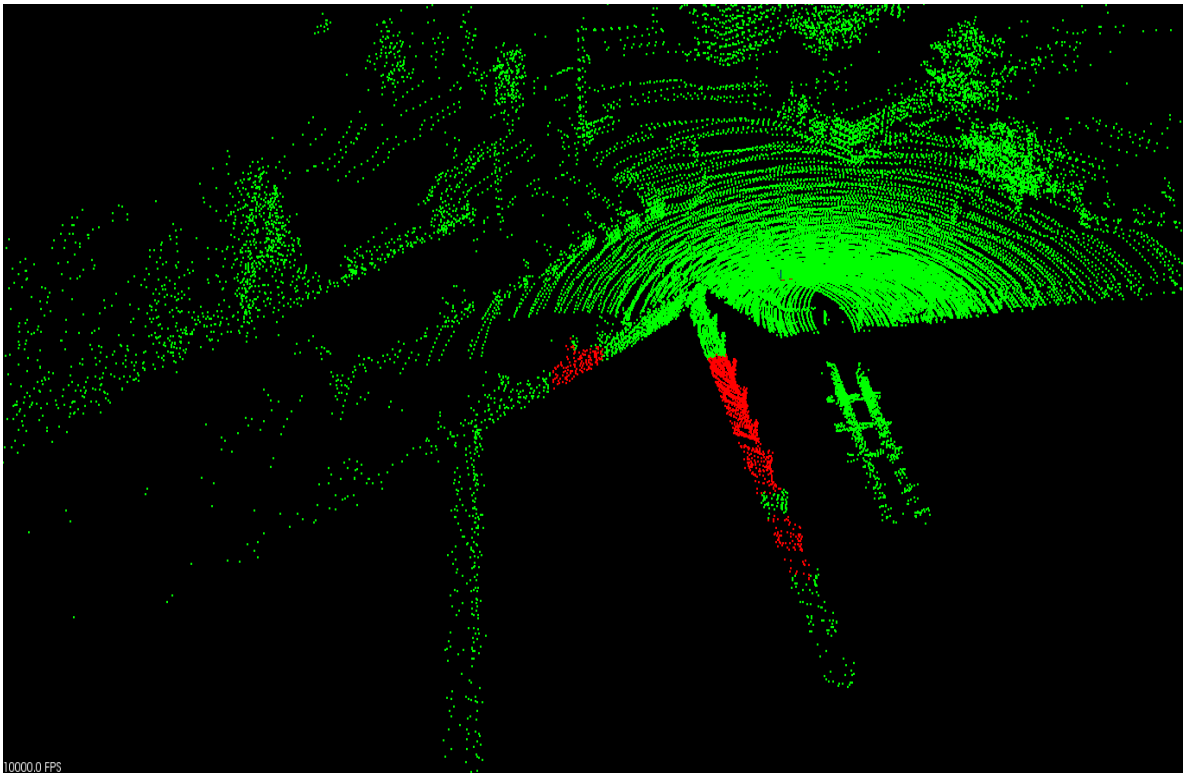

Figure 26: Part of the seawall identified as a false positive

The assumption called for the docks to be long, narrow, and skinny. At this point in the algorithm, a single piling does not represent a dock in the sense that the point cloud does not meet the previously mentioned descriptors. In order to check if the dock is long, narrow, and skinny, the output point cloud is clustered using parameters where each cluster represents an object, whether it be the dock, or a single piling. Eigenvalues are then calculated for each cluster. The ratio of the long eigenvalue over the short eigenvalue is determined. This ratio represents the length to width ratio of the point cloud. It was determined that the ratio should be at least two, meaning that the dock location should be twice as long as it is wide. However, sometimes objects are small and really skinny, such as the sea wall in Figure 26, which would produce a high ratio. For this reason, the small eigenvalue had to be larger than a certain value. This ensures that a cluster's width is actually wide enough to be a dock. Once each cluster has gone through this process, the ones that meet the thresholds are pushed through and identified as docks; likewise, the clusters that do not meet the thresholds are discarded. Figure 27 below exhibits the final output with the identified dock in red.

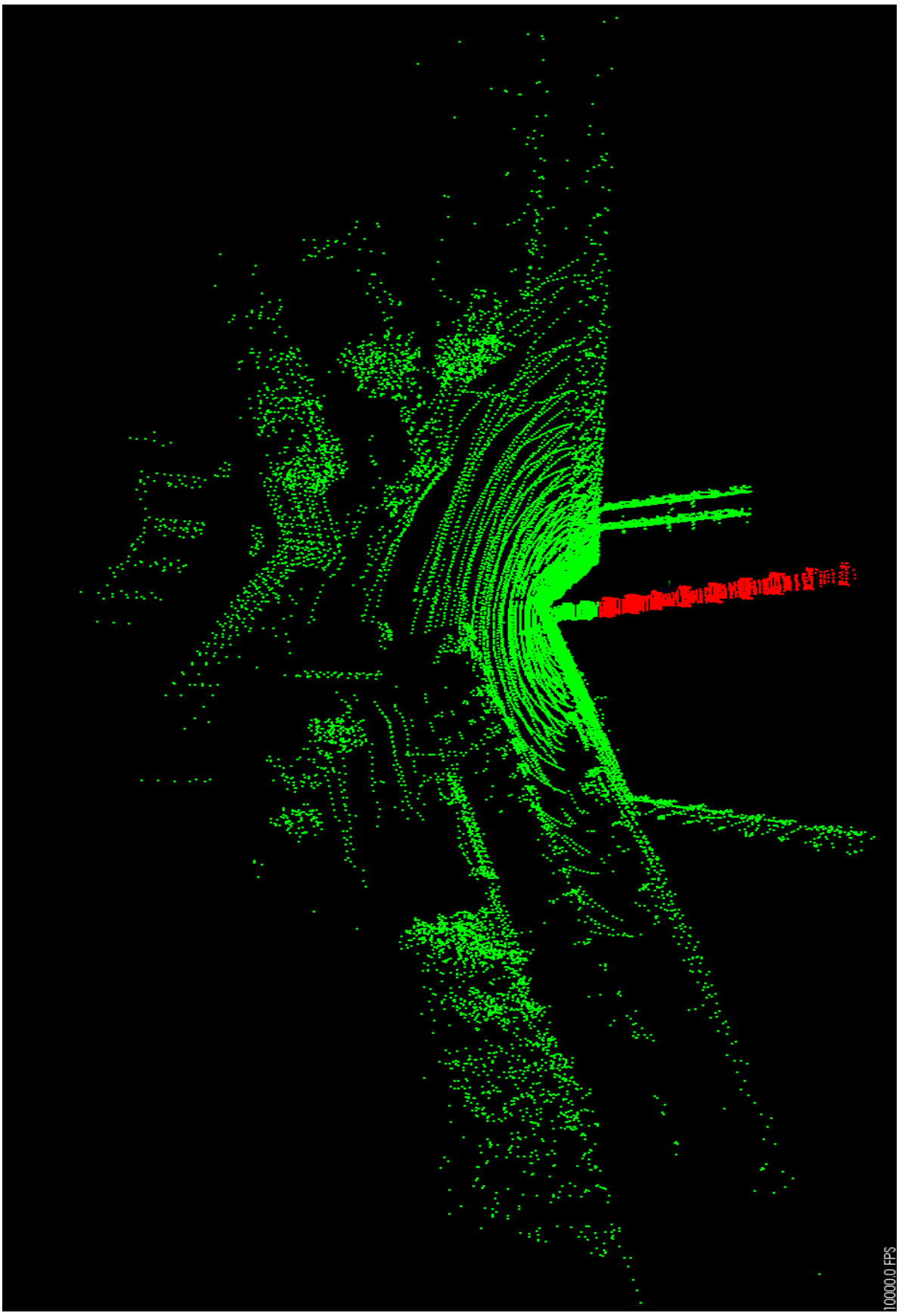

Figure 27: Shows the original point cloud with the dock identified in red

7 Results

From the previous figure, it is seen that identifying a suitable dock is possible; however, this scan is not a realistic scan. The data for the previous scan was collected while standing on top of the target dock; this is the reason why the laser scans are perpendicular with the length of the dock. Collecting data on top of the dock allowed for a series of very clean scans which resulted in a clean dense point cloud once all were transformed and concatenated. While this scan may not represent the ideal test scan for the applicability of an autonomous surface vessel, limitations in the algorithm can still be found. In Figure 27, it can be seen that the portion of dock close to the shore was not identified. The reason for this originates in the shore removal step and the morphological steps. When the binary image of the point cloud is originally dilated in order to fill in the gaps in the middle of objects, the shore connects with the dock and essentially extends the shore the length of the structuring element. In any scan, if the dock is not perfectly perpendicular with the shore or in a concave shape close enough for the structuring element to combine the two, part of the base of the dock will not be identified. However, while this is important, it only means that a targeted dock must protrude away from the shore.

As just stated, the first scan was not a realistic scan to prove this algorithm's applicability to autonomous surface vessels. The next step was to test the algorithm on a scan collected from another angle. Figure 28 shows an image of where the data was collected for this scan. Notice how the line is perpendicular with the dock which provides a new perspective for the algorithm. Figure 29, previously seen as Figure 26, shows the results of the algorithm used on this scan.

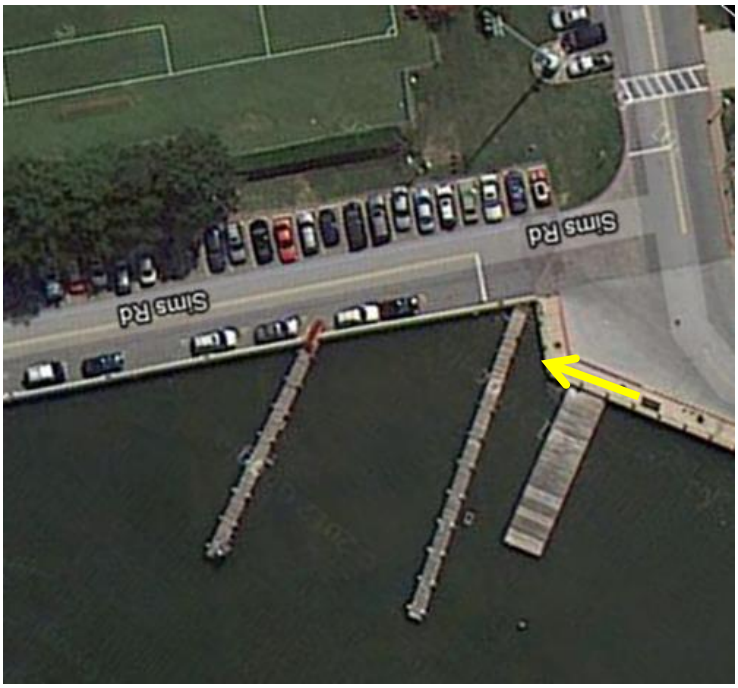

Figure 28: Data collected for Figure 29 along the yellow arrow

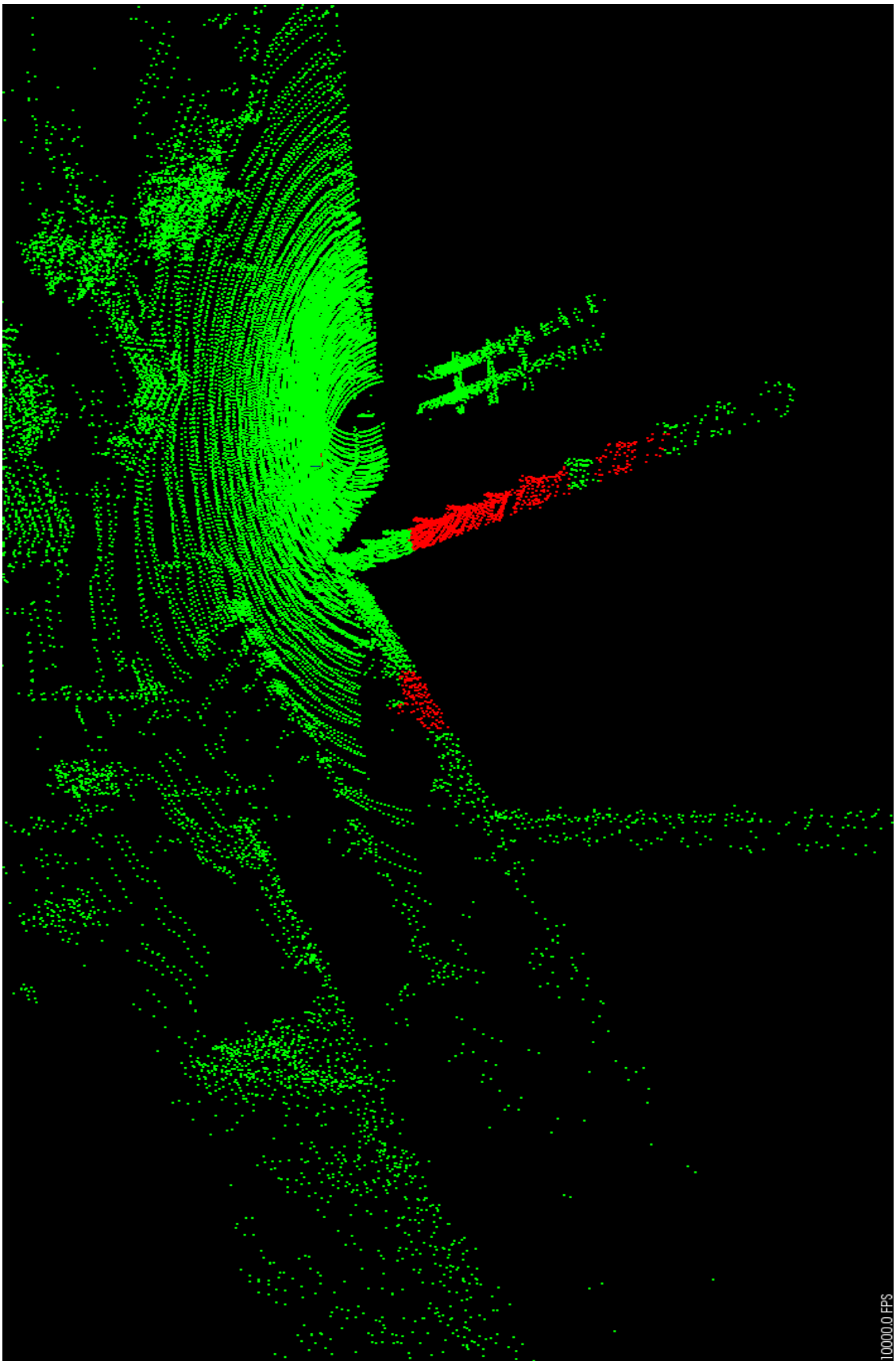

Figure 29: Shows identification of dock wheredata points are dense enough and a false positive along the sea wall

There are many points of interest in Figure 29. First note that the algorithm did identify the dock in some places. However, it is important to realize why the dock was not identified in its entirety. All the pilings were not detected. This problem arises from there not being enough points hitting all the pilings. Since the data was collected perpendicular with the dock along one line, the scan provides very dense data in the location in front of the sensor; however, the sensor failed to collect enough information about the end of the dock to identify anything as a piling. One of the most notable features in Figure 29 is that part of the seawall was identified as a dock. The reason that it is identified is that through the morphological and clustering process, the algorithm believed that there was a piling. Also, since there is a shadow area behind the cluster, the shore removal step did not remove the area as part of the shore. However, the image presented in Figure 29 was captured before the thresholds that evaluate each identified cluster were inserted into the algorithm. By inserting the thresholds, this false positive is eliminated as a suitable docking location. The same seen can be seen below in Figure 30 with the false positive eliminated.

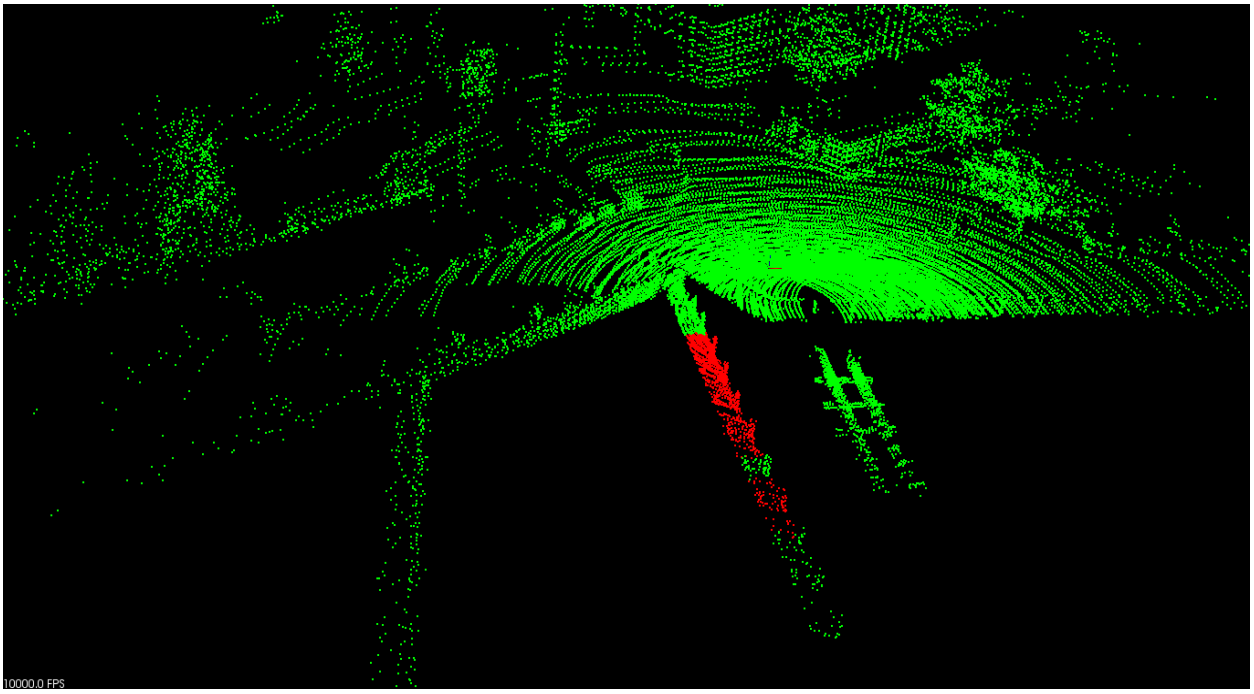

Figure 30: Algortihm output with false positive removed

The next scan that was tested was a scan whose data was collected by moving the cart along the dock that is parallel with the target dock, see Figure 31. The previous scan proved that a dock could be identified when the sensor collects data from the side. The goal of this scan was to try to identify the whole dock by collecting data the whole length of the target dock. As seen in Figure 32, most of the dock was identified; however, there were a few pilings missed. It should be noticed that the bottom dock is not whole because not enough sensor scans were concatenated to fill the void of where the sensor area under the LiDAR overlapped. The other positive aspect from this scan is the fact that parts of both docks were identified. Even though the two docks are similar, they are not the same dimensions. This is the first evidence of this algorithm identifying another dock besides the original targeted dock.

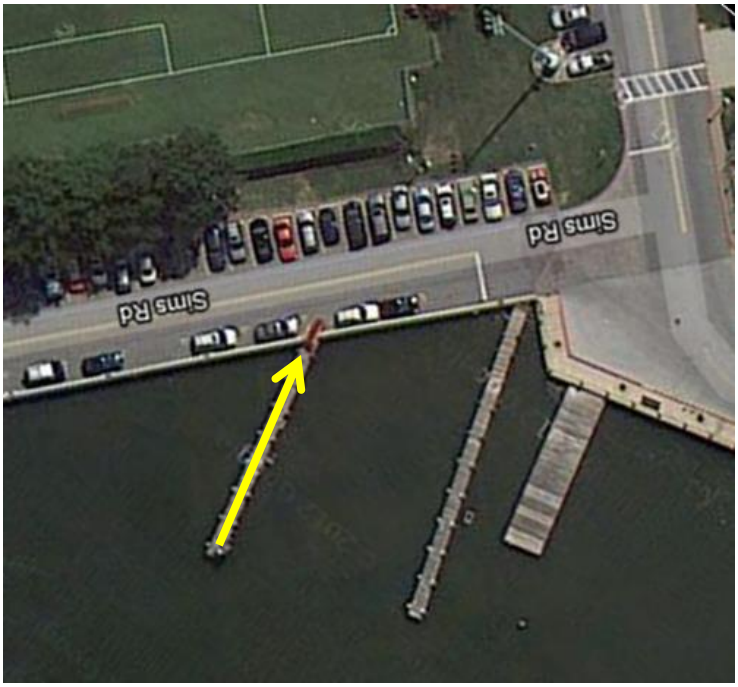

Figure 31: Data collected along the arrow
Imagery ©2013 U.S. Geological Survey. Map data ©2013 Google

Another issue in Figure 32 is the fact that there are a lot of points on the water. These data points are not errors in the sensor, they are caused by debris in the water, see Figure 33 for a closer view. Before this series of scans were obtained, the dock was piled high with ocean debris, such as little branches. In order for the cart to sit level with the dock, the debris was pushed into the water. These pieces of debris were then picked up on the scan. The relevance of this is that in the algorithm each data point creates a pixel in the binary images that is one square meter. Now, the pixel size can be changed to represent any size; however, if it is changed, computation time has to be taken into account, along with the resizing of the morphological structuring elements in use. A singular piece of debris is not enough to alter the algorithm; yet, a few pieces of debris around the dock can cause the dock to appear thicker than it really is in the binary images. This can lead the morphology to not eliminate the dock from the shore.

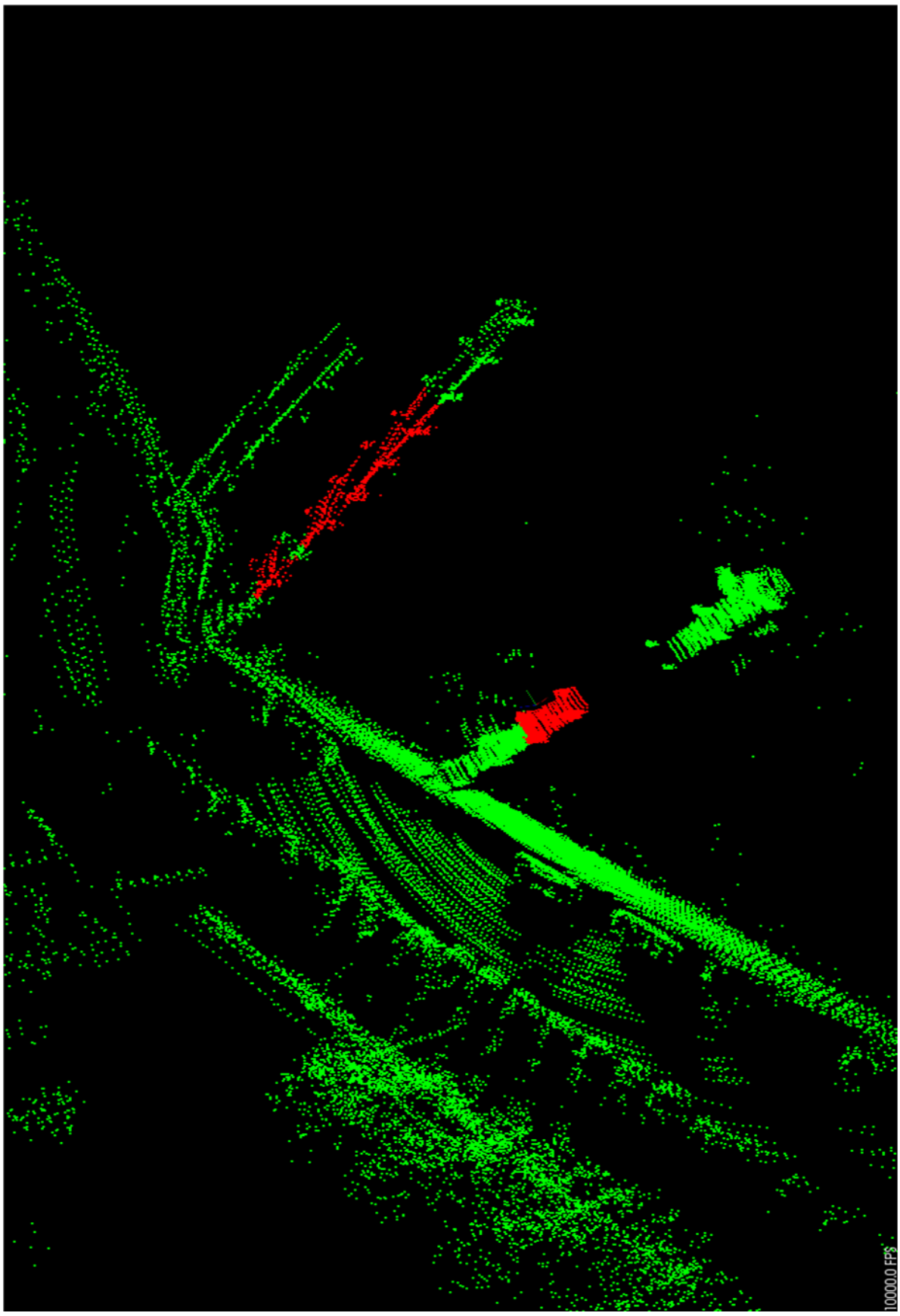

Figure 32: Dock identification using data collected parallel to target dock

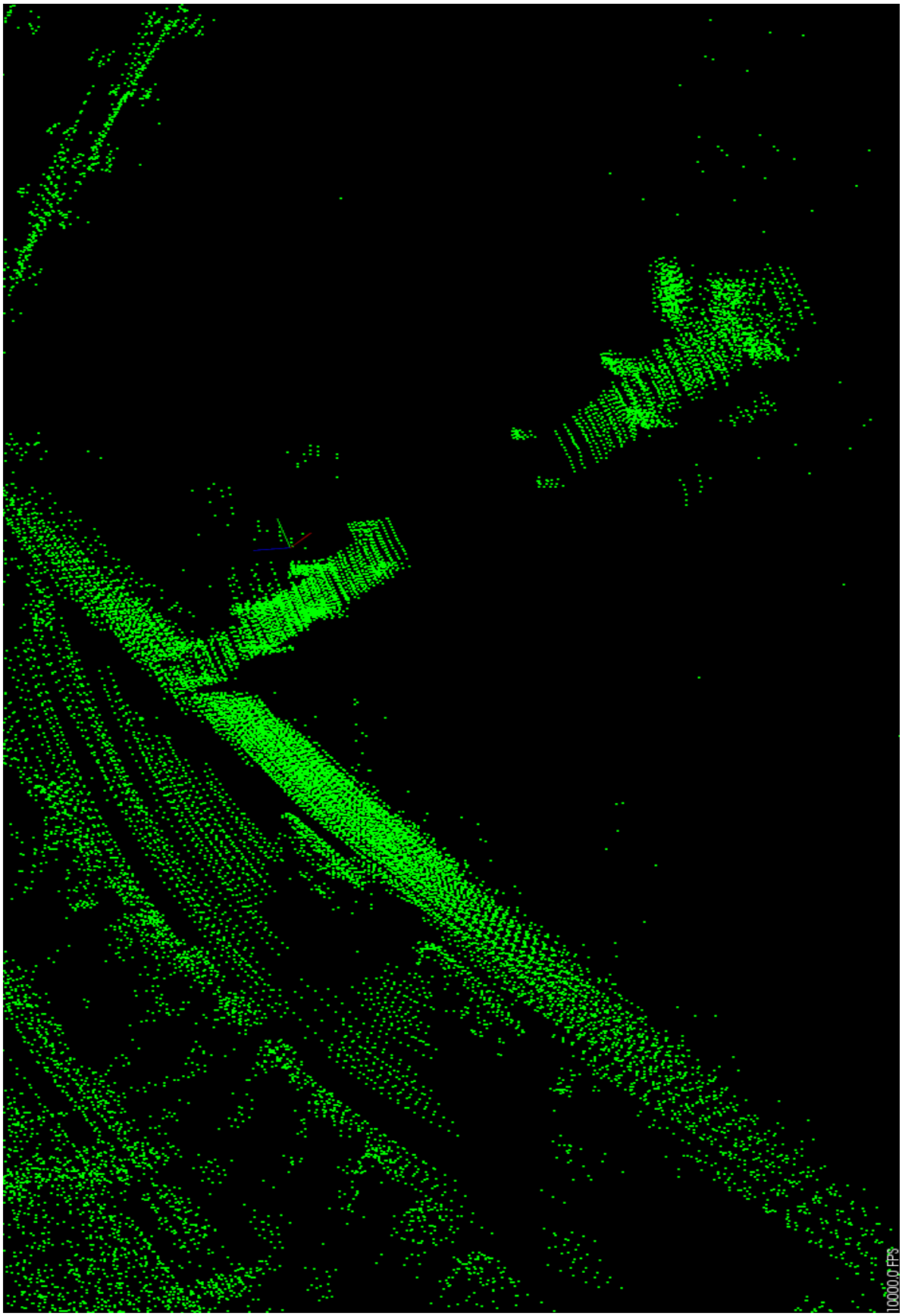

Figure 33: Image of sensor reading debris on the surface of the water

All of the previous scans were obtained from the same dock at different angles. One of the last scans that the algorithm was tested on was a scan of a foreign area. The new target dock was chosen because, while different from the original target dock, it contained a lot of the same features and was of similar size. Figure 34 is an image showing where the data was collected and the new target dock, while Figure 35 is an image of the target dock.

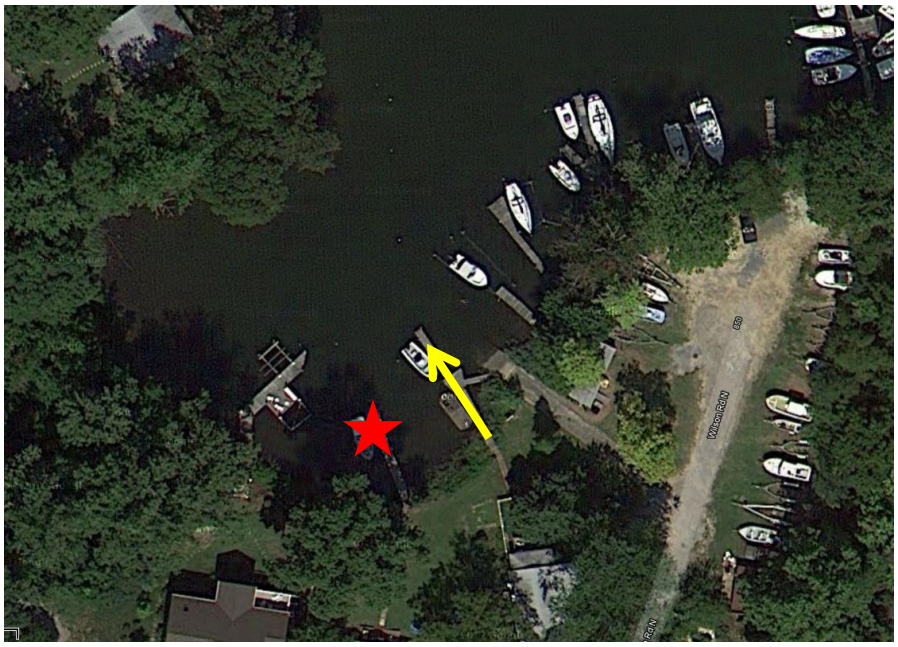

Figure 34: Yellow arrow represents where the data was collected while the red star indicates the target dock
Imagery ©2013 U.S. Geological Survey. Map data ©2013 Google

Figure 35: Image of the new target dock

The purpose of this scan was to test the algorithm's robustness, meaning that it could be used on any dock with similiar dimensions of the original target dock as opposed to only identifying the target dock. Figure 36 illustrates that the algorithm identified two docks. The end pilings on the upper dock in Figure 36 were not identified. There are a couple possibilities for why this happened; the first is that the pilings may be too tall to be recognized. These two pilings were a foot and a half taller than the other pilings. Since the pilings were taller, the algorithm may have ignored them because all of the data collected on known pilings for the probability density function was gathered off of shorter pilings. The more likely reason that these pilings were not identified is that there were a lot of spurious returns collected underneath the dock. Unlike the last scan where debris caused the data points, in this case, the points are caused by some other effect. The theory with this dock is that the water is less than a foot deep and there are many rocks under the dock, so it is believed that the lasers were not affected by that shallow of water. These spurious returns can be seen underneath the dock in figure 37. These points, more likely than not, were clustered with the two end pilings. When the features were calculated for each cluster in the piling identification step, these extraneous data points would have altered the values. This scan shows that this algorithm may not work properly when the water beneath the dock is shallow enough for the lasers to not refract. The depth of the water beneath the docking locations was assumed to have no effect on the return of the scans; however, this scan shows evidence that this cannot be assumed.

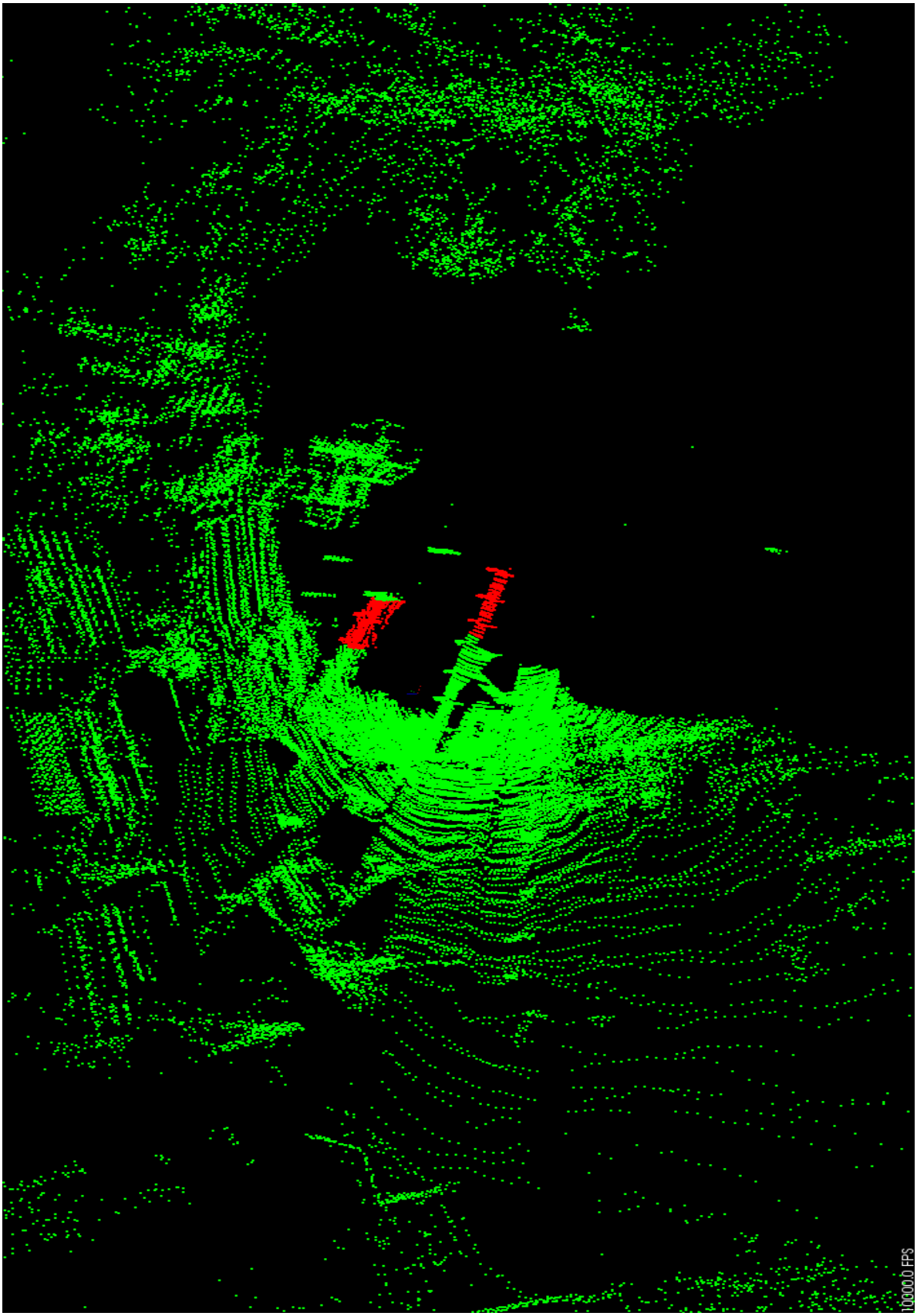

Figure 36: Scan of a new area with algorithm able to identify two docks

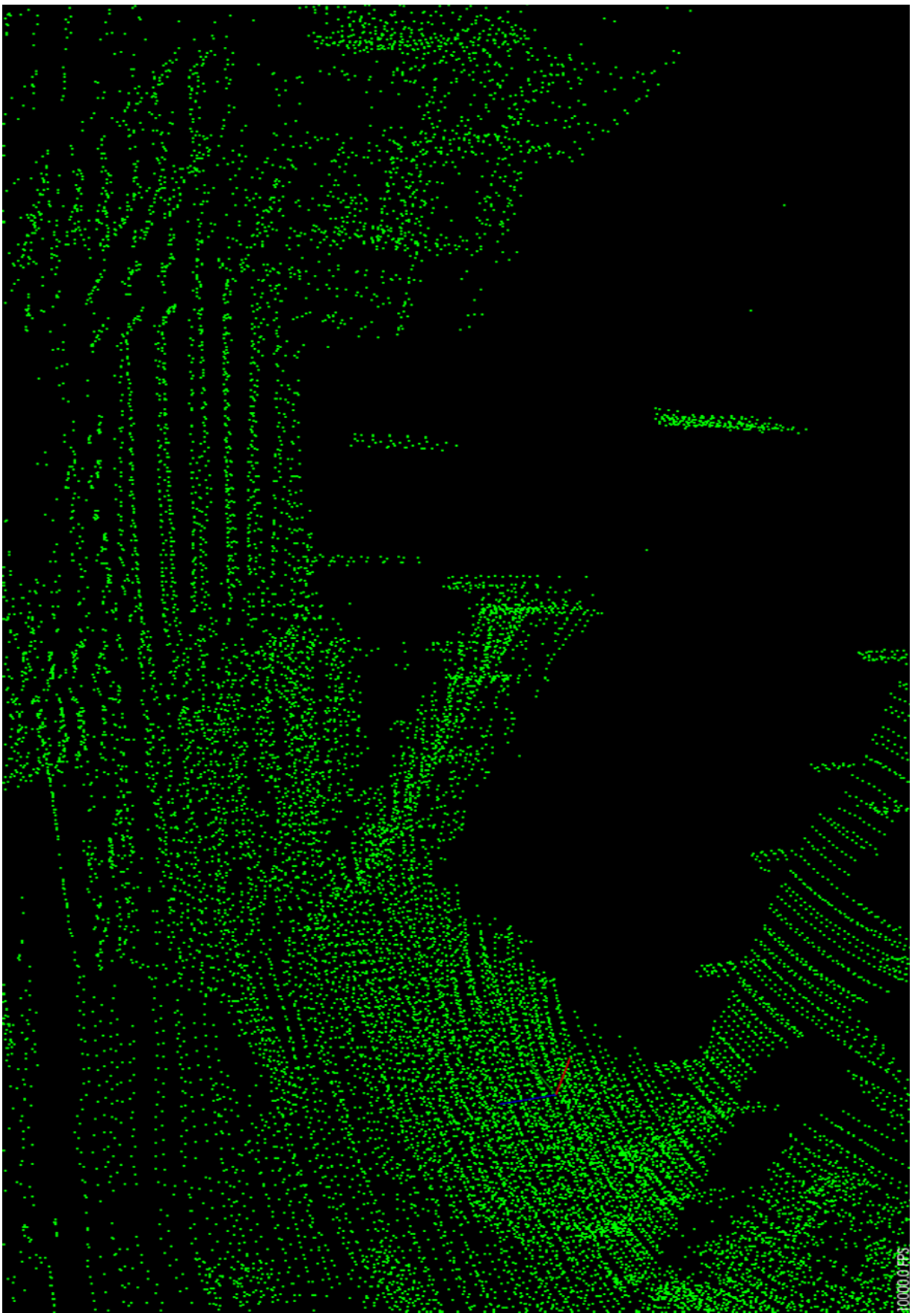

Figure 37: This image illustrates the spurious returns that were identified under the dock during this scan

8 Conclusion and Future Work

The previous scans show that it is possible to isolate a dock from dense point clouds. The algorithm works well when there are enough data points to represent the whole dock. Since the registration process was long and arduous, one way to improve the testing of the algorithm is to get an algorithm which uses the LiDAR to register multiple point clouds as it collects them. This process would allow for faster collection of dense point clouds and would be the first step to implementing the algorithm in real time. Research is currently being done on SLAM (Simultaneous Localization and Mapping) algorithms which do just that, map and register unknown environments. If an autonomous surface vessel were to map an unknown area, this algorithm could be used once the vessel was done mapping to identify docking locations. If it were possible to collect more dense scans faster, the algorithm could have been more thoroughly tested.

Having more scans would allow for more thorough supervised training of piling clusters during the piling identification step. The positive identification of pilings is one of the largest limiting factors in the algorithm for identifying docks. In the current algorithm, the only pilings included into the training data were pilings from the target dock. If known pilings from other docks, which varied in diameter and height, were introduced to the traing data, the Gaussian PDF would allow a larger variance in piling size. This means that if a dock contains pilings that are taller, shorter, wider or thicker, than the original target dock's pilings, the training data would be robost enough to still identify those pilings.

While implementing a global registration process would be helpful, it most likely will reveal some shortcomings with the current algorithm that would need to be improved. In theory, a SLAM algorithm on an autonomus surface vessel would be able to map a whole harbor, where there are multiple docks of varying shapes and sizes. The algorithm should be able to identify each dock; however, this will most likely not happen. If a point cloud map of a harbor is collected, with multiple docks varying in shape and size, the algorithm would then remove the shore, and then attempt to find the plane. However, the way the current algorithm is written is that RANSAC will find the best fitting plane from the point cloud. In the case of testing this algorithm, this has been acceptable since the target docks are the only planar surface in the scan, and in each case with multiple docks, the docks' planar surfaces have been the same height. In order to ensure accuracy when using this algorithm on multiple docks, each dock should be first clustered and then have the planar surface identified within each cluster. This will ensure that RANSAC evaluates the best fitting plane for each individual dock instead of finding a single plane for the whole point cloud..

From the output of the algorithm, it is possible to get specific docking information, such as the centroid of the dock, location of individual identified pilings, or the bounding box coordinates for each dock. From this information, a program could be written to enable an autonomous surface vessel to dock at the identified docking location. Also, an algorithm could be written to survey the area surrounding the dock for obstacles and this information could be used to determine the best docking location for a surface vessel.

Overall, the algorithm does work for its specific objective; however, future work should initially focus on expanding the training data for pilings while continuing to test the algorithm on docks other than the original target dock.

9 Bibliography

Besl, Paul J. and Nei. D. McKay. “A method for registration of 3-D shapes”. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

Bohren, Jonathon., Tully Foote, Jim Keller, Alex Kushleyev, Daniel Lee, Alex Stewart, and Paul Vernaza. “Little Ben: The Ben Franklin Racing Team’s Entry in the 2007 DARPA Urban Challenge”. *Journal of Robotics*, 25(9), 598-614 (2008).

Bradski, Gary and Adrian Kaehler. *Learning OpenCV: ComputerVision with the OpenCV Library*. Sebastopol: O’Reilly, 2008.

Department of the Navy, *The Navy Unmanned Surface Vessel (USV) Master Plan.* Available at http://www.navy.mil/navydata/technology/usvmppr.pdf (Accessed 21 December 2011).

Feng, His-Yung. and Hao Song. **“**A Global Clustering Approach to Point Cloud Simplification with a Specified Data Reduction Ratio.” *Computer Aided Design*, 40(3): 281-292, March 2008.

Fishler, Martin A. and Robert C. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography,” *Communications of the ACM*, June 1981, 24(6):381-395.

Gonzalez, Rafael C., Richard E. Woods, and Steven L. Eddins. *Digital Image Processing Using MATLAB*. Natick: Gatesmark, 2009.

Leonard, John., Jonathon How, Seth Teller, Mitch Berger, Stefan Campbell, Gaston Fiore, Luke Fletcher, Emilio Frazzoli, Albert Huang, Sertac Karaman, Oliver Koch, Yoshiaki Kuwata, David Moore, Edwin Olson, Steve Peters, Justin Teo, Robert Truax, and Matthew Walter. “A Perception-Driven Autonomous Vehicle”. *Journal of Robotics*, 25(10), 727-774 (2008).

“Maritime Applications of LiDAR Sensors for Obstacle Avoidance and Navigation”. Velodyne LiDAR. 2009. 14 December 2011. <http://velodynelidar.com/lidar/products/white_paper/HDL%20white%20pages_Maritime_lowres.pdf >.

Point Cloud Library. http://Pointclouds.org (accessed August 27, 2012).

Rasu, Radu B. Nico Blodow, and Michael Beetz,”Fast Point Feature Histograms (FPFH) for 3D Registration” *IEEE International Conference on Robotics and Automation 2009*. 3212-3217, 2009.

Schnabel, R., R.Wahl and R. Klein. “Shape Detection in Point Clouds”. *Computer Graphics Technical Reports.* Volume 2, 2006.

Velodyne LiDAR, Inc. *User’s Manual and Programming Guide: HDL-32E High Definition LiDAR Sensor*. Morgan Hill: Velodyne LiDAR, Inc, 2011.

10 Appendices

Appendix A: Parameters Values and Transformation Matrix Examples

Align point cloud 5 (Location 310cm.pcd) to point cloud 4 (Location 610cm.pcd)

SAC-IA Parameters

Number of Samples: 10
Normal Radius: 2.4
Feature Radius: 2.4
Minimum Sample Distance: 5.0
Maximum Correspondence Distance: .001
Maximum of Iterations: 500

ICP Parameters

Max Correspondence Distance: 3.5
Transformation Epsilon: .000005
Maximum of Iterations: 200

$$T_4^5 = \begin{bmatrix} .997 & -.074 & .021 & -2.943 \\ .074 & .997 & .006 & 1 \\ -.021 & -.004 & 1 & -.2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Align point cloud 4 (Location 610cm.pcd) to point cloud 3 (Location 910cm.pcd)

SAC-IA Parameters

Number of Samples: 10
Normal Radius: 2.3
Feature Radius: 2.3
Minimum Sample Distance: 5.0
Maximum Correspondence Distance: .001
Maximum of Iterations: 500

ICP Parameters

Max Correspondence Distance: 4.5
Transformation Epsilon: .000005
Maximum of Iterations: 200

$$T_3^4 = \begin{bmatrix} .999 & .033 & -.011 & -3.126 \\ -.033 & .999 & -.006 & .021 \\ .01 & .007 & 1 & .1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Align point cloud 3 (Location 910cm.pcd) to point cloud 2 (Location 1210cm.pcd)

SAC-IA Parameters

Number of Samples: 10
Normal Radius: 2.6
Feature Radius: 2.6
Minimum Sample Distance: 5.0
Maximum Correspondence Distance: .001
Maximum of Iterations: 500

ICP Parameters

Max Correspondence Distance: 4
Transformation Epsilon: .00001
Maximum of Iterations: 200

$$T_2^3 = \begin{bmatrix} 1 & -.015 & .015 & -2.892 \\ .015 & 1 & -.002 & .25 \\ -.015 & .002 & 1 & .003 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Align point cloud 2 (Location 1210cm.pcd) to point cloud 1 (Location 1510cm.pcd)

SAC-IA Parameters

Number of Samples: 10
Normal Radius: 3.0
Feature Radius: 3.1
Minimum Sample Distance: 5.0
Maximum Correspondence Distance: .001
Maximum of Iterations: 500

ICP Parameters

Max Correspondence Distance: 5
Transformation Epsilon: .00001
Maximum of Iterations: 200

$$T_1^2 = \begin{bmatrix} 1 & .023 & .004 & -3.005 \\ -.023 & 1 & .009 & .135 \\ -.004 & -.009 & 1 & .027 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Align point cloud 1 (Location 1510cm.pcd) to point cloud 0 (Location 1810cm.pcd)

SAC-IA Parameters

Number of Samples: 10
Normal Radius: 3.0
Feature Radius: 3.1
Minimum Sample Distance: 5.0
Maximum Correspondence Distance: .001
Maximum of Iterations: 500

ICP Parameters

Max Correspondence Distance: 5
Transformation Epsilon: .00001
Maximum of Iterations: 200

$$T_0^1 = \begin{bmatrix} .999 & .032 & -.012 & -3.068 \\ -.032 & .999 & -.021 & .158 \\ .011 & .021 & 1 & -.057 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Appendix B: Code

Registration Code (Part1)

```
#include <iostream>
#include <boost/thread/thread.hpp>
#include <pcl/common/common_headers.h>
#include <pcl/features/normal_3d.h>
#include <pcl/io/pcd_io.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/console/parse.h>
#include <pcl/point_types.h>
#include <pcl/registration/icp.h>
#include <limits>
#include <fstream>
#include <vector>
#include <Eigen/Core>
#include <pcl/point_cloud.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/fpfh.h>
#include <pcl/registration/ia_ransac.h>


boost::shared_ptr<pcl::visualization::PCLVisualizer>
simpleVis (pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_in,
pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_out)
{


  // --------------------------------------------
  // -----Open 3D viewer and add point cloud-----
  // --------------------------------------------
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new
pcl::visualization::PCLVisualizer ("3D Viewer"));// creates object viewer
  viewer->setBackgroundColor (0, 0, 0);

  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_in_color(cloud_in, 0, 0, 255);// sets cloud_in, which will be cloud out
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_out_color(cloud_out, 255, 255, 0); // sets cloud_out to blue
  viewer->addPointCloud<pcl::PointXYZ> (cloud_in, cloud_in_color, "cloud_in"); //adds
point cloud and defines it as cloud_in
  viewer->addPointCloud<pcl::PointXYZ> (cloud_out, cloud_out_color, "cloud_out"); //adds
point cloud and defines it as cloud_out
  viewer->setPointCloudRenderingProperties
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 1, "cloud_in");
  viewer->addCoordinateSystem (1.0);
  viewer->initCameraParameters ();
  return (viewer);
}

boost::shared_ptr<pcl::visualization::PCLVisualizer>
```

```
simpleVis (pcl::PointCloud<pcl::PointXYZ>::ConstPtr final,
pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_in,
pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_out,
pcl::PointCloud<pcl::PointXYZ>::ConstPtr icp)
{
// --------------------------------------------
  // -----Open 3D viewer and add point cloud-----
  // --------------------------------------------
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer2 (new
pcl::visualization::PCLVisualizer ("3D Viewer"));// creates object viewer
  viewer2->setBackgroundColor (0, 0, 0);

  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> final_color(final, 255,
0, 0);// sets final to red
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_in_color(cloud_in, 0, 255, 0);// cloud in is green
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_out_color(cloud_out, 0, 0, 255); // sets cloud_out to blue
   pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> cloud_icp_color(icp,
255, 255, 0); // colors icp yellow
  viewer2->addPointCloud<pcl::PointXYZ> (final, final_color, "final"); //adds point cloud
and defines it as final
  viewer2->addPointCloud<pcl::PointXYZ> (cloud_in, cloud_in_color, "cloud_in"); //adds
point cloud and defines it as cloud_in
  viewer2->addPointCloud<pcl::PointXYZ> (cloud_out, cloud_out_color, "cloud_out"); //adds
point cloud and defines it as cloud_out
  viewer2->addPointCloud<pcl::PointXYZ> (icp, cloud_icp_color, "icp");
  viewer2->setPointCloudRenderingProperties
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 1, "final");
    viewer2->addCoordinateSystem (1.0);
  viewer2->initCameraParameters ();
  return (viewer2);
}

class FeatureCloud
{
  public:
    // A bit of shorthand
    typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
    typedef pcl::PointCloud<pcl::Normal> SurfaceNormals;
    typedef pcl::PointCloud<pcl::FPFHSignature33> LocalFeatures;
    typedef pcl::search::KdTree<pcl::PointXYZ> SearchMethod;

    FeatureCloud () :
      search_method_xyz_ (new SearchMethod),
      normal_radius_ (1.4f),
      feature_radius_ (1.5f)
    {}

    ~FeatureCloud () {}

    // Process the given cloud
    void
    setInputCloud (PointCloud::Ptr xyz)
    {
      xyz_ = xyz;
      processInput ();
    }
```

```
  // Load and process the cloud in the given PCD file
  void
  loadInputCloud (const std::string &pcd_file)
  {
    xyz_ = PointCloud::Ptr (new PointCloud);
    pcl::io::loadPCDFile (pcd_file, *xyz_);
    processInput ();
  }

  // Get a pointer to the cloud 3D points
  PointCloud::Ptr
  getPointCloud () const
  {
    return (xyz_);
  }

  // Get a pointer to the cloud of 3D surface normals
  SurfaceNormals::Ptr
  getSurfaceNormals () const
  {
    return (normals_);
  }

  // Get a pointer to the cloud of feature descriptors
  LocalFeatures::Ptr
  getLocalFeatures () const
  {
    return (features_);
  }

protected:
  // Compute the surface normals and local features
  void
  processInput ()
  {
    computeSurfaceNormals ();
    computeLocalFeatures ();
  }

  // Compute the surface normals
  void
  computeSurfaceNormals ()
  {
    normals_ = SurfaceNormals::Ptr (new SurfaceNormals);

    pcl::NormalEstimation<pcl::PointXYZ, pcl::Normal> norm_est;
    norm_est.setInputCloud (xyz_);
    norm_est.setSearchMethod (search_method_xyz_);
    norm_est.setRadiusSearch (normal_radius_);
    norm_est.compute (*normals_);
  }

  // Compute the local feature descriptors
  void
  computeLocalFeatures ()
  {
    features_ = LocalFeatures::Ptr (new LocalFeatures);
```

```
      pcl::FPFHEstimation<pcl::PointXYZ, pcl::Normal, pcl::FPFHSignature33> fpfh_est;
      fpfh_est.setInputCloud (xyz_);
      fpfh_est.setInputNormals (normals_);
      fpfh_est.setSearchMethod (search_method_xyz_);
      fpfh_est.setRadiusSearch (feature_radius_);
      fpfh_est.compute (*features_);
    }

  private:
    // Point cloud data
    PointCloud::Ptr xyz_;
    SurfaceNormals::Ptr normals_;
    LocalFeatures::Ptr features_;
    SearchMethod::Ptr search_method_xyz_;

    // Parameters
    float normal_radius_;
    float feature_radius_;
};

class TemplateAlignment
{
  public:

    // A struct for storing alignment results
    struct Result
    {
      float fitness_score;
      Eigen::Matrix4f final_transformation;
      EIGEN_MAKE_ALIGNED_OPERATOR_NEW
    };

    TemplateAlignment () :

      min_sample_distance_ (3.0f),
      max_correspondence_distance_ (.001f),
      nr_iterations_ (500)
    {
      // Intialize the parameters in the Sample Consensus Intial Alignment (SAC-IA)
algorithm
              sac_ia_.setNumberOfSamples(5);
      sac_ia_.setMinSampleDistance (min_sample_distance_);
      sac_ia_.setMaxCorrespondenceDistance (max_correspondence_distance_);
      sac_ia_.setMaximumIterations (nr_iterations_);
    }

    ~TemplateAlignment () {}

    // Set the given cloud as the target to which the templates will be aligned
    void
    setTargetCloud (FeatureCloud &target_cloud)
    {
      target_ = target_cloud;
      sac_ia_.setInputTarget (target_cloud.getPointCloud ());
      sac_ia_.setTargetFeatures (target_cloud.getLocalFeatures ());
    }
```

```
    // Add the given cloud to the list of template clouds
    void
    addTemplateCloud (FeatureCloud &template_cloud)
    {
      templates_.push_back (template_cloud);
    }

    // Align the given template cloud to the target specified by setTargetCloud ()
    void
    align (FeatureCloud &template_cloud, TemplateAlignment::Result &result)
    {
      sac_ia_.setInputCloud (template_cloud.getPointCloud ());
      sac_ia_.setSourceFeatures (template_cloud.getLocalFeatures ());

      pcl::PointCloud<pcl::PointXYZ> registration_output;
      sac_ia_.align (registration_output);

      result.fitness_score = (float) sac_ia_.getFitnessScore
(max_correspondence_distance_);
      result.final_transformation = sac_ia_.getFinalTransformation ();
    }

    // Align all of template clouds set by addTemplateCloud to the target specified by
setTargetCloud ()
    void
    alignAll (std::vector<TemplateAlignment::Result, Eigen::aligned_allocator<Result> >
&results)
    {
      results.resize (templates_.size ());
      for (size_t i = 0; i < templates_.size (); ++i)
      {
        align (templates_[i], results[i]);
      }
    }

    // Align all of template clouds to the target cloud to find the one with best
alignment score
    int
    findBestAlignment (TemplateAlignment::Result &result)
    {
      // Align all of the templates to the target cloud
      std::vector<Result, Eigen::aligned_allocator<Result> > results;
      alignAll (results);

      // Find the template with the best (lowest) fitness score
      float lowest_score = std::numeric_limits<float>::infinity ();
      int best_template = 0;
      for (size_t i = 0; i < results.size (); ++i)
      {
        const Result &r = results[i];
        if (r.fitness_score < lowest_score)
        {
          lowest_score = r.fitness_score;
          best_template = (int) i;
        }
      }

      // Output the best alignment
```

```
      result = results[best_template];
      return (best_template);
    }

  private:
    // A list of template clouds and the target to which they will be aligned
    std::vector<FeatureCloud> templates_;
    FeatureCloud target_;

    // The Sample Consensus Initial Alignment (SAC-IA) registration routine and its
parameters
    pcl::SampleConsensusInitialAlignment<pcl::PointXYZ, pcl::PointXYZ,
pcl::FPFHSignature33> sac_ia_;
    float min_sample_distance_;
    float max_correspondence_distance_;
    int nr_iterations_;
};


int
main (int argc, char** argv)
{
  //If need to filter point clouds, un-comment the declarations of cloud_in_unfiltered1 &
2
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_in_unfiltered1 (new
pcl::PointCloud<pcl::PointXYZ>); // cloud_in will hopefully be base of registration
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_in_unfiltered2 (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_in_unfiltered3 (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_out (new pcl::PointCloud<pcl::PointXYZ>); //
cloud_out will be registered to cloud
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_in (new pcl::PointCloud<pcl::PointXYZ>);

  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_out_unfiltered1 (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_out_unfiltered2 (new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_out_unfiltered3 (new
pcl::PointCloud<pcl::PointXYZ>);

  pcl::io::loadPCDFile("IncrementalDockData\\Line4 750cm.pcd", *cloud_in_unfiltered1);
  pcl::PassThrough<pcl::PointXYZ> pass;
  pass.setInputCloud (cloud_in_unfiltered1);
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (-5.0, 0.0);
  pass.filter(*cloud_in_unfiltered2);
  pass.setInputCloud(cloud_in_unfiltered2);
  pass.setFilterFieldName ("y");
  pass.setFilterLimits (-30.0, 10.0);
  pass.filter(*cloud_in_unfiltered3);
  pass.setInputCloud(cloud_in_unfiltered3);
  pass.setFilterFieldName ("x");
  pass.setFilterLimits (-30.0, 10.0);
  pass.filter(*cloud_in);


  FeatureCloud template_cloud;
```

```
  template_cloud.setInputCloud(cloud_in);

  pcl::io::loadPCDFile ("IncrementalDockData\\Line4 900cm.pcd", *cloud_out_unfiltered1);
   pass.setInputCloud (cloud_out_unfiltered1);
  pass.setFilterFieldName ("z");
  pass.setFilterLimits (-5.0, 0.0);
  pass.filter(*cloud_out_unfiltered2);
  pass.setInputCloud(cloud_out_unfiltered2);
  pass.setFilterFieldName ("y");
  pass.setFilterLimits (-30.0, 10.0);
  pass.filter(*cloud_out_unfiltered3);
  pass.setInputCloud(cloud_out_unfiltered3);
  pass.setFilterFieldName ("x");
  pass.setFilterLimits (-30.0, 10.0);
  pass.filter(*cloud_out);

  // Assign to the target FeatureCloud
  FeatureCloud target_cloud;
  target_cloud.setInputCloud (cloud_out);

  // Set the TemplateAlignment inputs
 TemplateAlignment template_align;
 // for (size_t i = 0; i < object_templates.size (); ++i)
  //{
  template_align.addTemplateCloud (template_cloud);
 // }
  template_align.setTargetCloud (target_cloud);

  // Find the best template alignment
  TemplateAlignment::Result best_alignment;
  int best_index = template_align.findBestAlignment (best_alignment);
  const FeatureCloud &best_template = template_cloud;

  // Print the alignment fitness score (values less than 0.00002 are good)
  printf ("Best fitness score: %f\n", best_alignment.fitness_score);

  // Print the rotation matrix and translation vector
  Eigen::Matrix3f rotation = best_alignment.final_transformation.block<3,3>(0, 0);
  Eigen::Vector3f translation = best_alignment.final_transformation.block<3,1>(0, 3);
  Eigen::Matrix4f SACIAMatrix = best_alignment.final_transformation;

  printf ("\n");
  printf ("    | %6.3f %6.3f %6.3f | \n", rotation (0,0), rotation (0,1), rotation
(0,2));
  printf ("R = | %6.3f %6.3f %6.3f | \n", rotation (1,0), rotation (1,1), rotation
(1,2));
  printf ("    | %6.3f %6.3f %6.3f | \n", rotation (2,0), rotation (2,1), rotation
(2,2));
  printf ("\n");
  printf ("t = < %0.3f, %0.3f, %0.3f >\n", translation (0), translation (1), translation
(2));

  // Save the aligned template for visualization
  pcl::PointCloud<pcl::PointXYZ> final;
  pcl::transformPointCloud (*best_template.getPointCloud (), final,
best_alignment.final_transformation);
  pcl::io::savePCDFile ("NewFile1.pcd", final);
  pcl::PointCloud<pcl::PointXYZ>::ConstPtr finalPtr ( &final);
```

```
  pcl::IterativeClosestPoint<pcl::PointXYZ, pcl::PointXYZ> icp;
  //icp.setRANSACOutlierRejectionThreshold();
  icp.setMaxCorrespondenceDistance(3.0);
  icp.setTransformationEpsilon (.00001);
  icp.setMaximumIterations (200);
  icp.setInputCloud(finalPtr);
  icp.setInputTarget(cloud_out);
  pcl::PointCloud<pcl::PointXYZ> ICPFinal;
  icp.align(ICPFinal);
  //icp.getFinalTransformation()>FinalMatrix;

  Eigen::Matrix4f rotationICP = icp.getFinalTransformation();
 // Eigen::Vector3f TranslationICP = icp.getFinalTransformation().block<3,1>(0, 3);
  printf("The fitness score is: %f\n", icp.getFitnessScore());

  printf ("\n");
  printf ("    | %6.3f %6.3f %6.3f %6.3f | \n", rotationICP (0,0), rotationICP (0,1),
rotationICP (0,2), rotationICP(0,3));
  printf ("R = | %6.3f %6.3f %6.3f %6.3f | \n", rotationICP (1,0), rotationICP (1,1),
rotationICP (1,2), rotationICP(1,3));
  printf ("    | %6.3f %6.3f %6.3f %6.3f | \n", rotationICP (2,0), rotationICP (2,1),
rotationICP (2,2), rotationICP(2,3));
  printf ("    | %6.3f %6.3f %6.3f %6.3f | \n", rotationICP (3,0), rotationICP (3,1),
rotationICP (3,2), rotationICP(3,3));
  printf ("\n");
  //Save ICPFinal as PCD File
  pcl::io::savePCDFile ("NewFile2.pcd", ICPFinal);
  pcl::PointCloud<pcl::PointXYZ>::ConstPtr ICPFinalPtr ( &ICPFinal);

  Eigen::Matrix4f FinalMatrix = SACIAMatrix * rotationICP;
  printf("Final Transformation Matrix\n");
  printf ("\n");
  printf ("    | %6.3f %6.3f %6.3f %6.3f | \n", FinalMatrix (0,0), FinalMatrix (0,1),
FinalMatrix (0,2), FinalMatrix(0,3));
  printf ("R = | %6.3f %6.3f %6.3f %6.3f | \n", FinalMatrix (1,0), FinalMatrix (1,1),
FinalMatrix (1,2), FinalMatrix(1,3));
  printf ("    | %6.3f %6.3f %6.3f %6.3f | \n", FinalMatrix (2,0), FinalMatrix (2,1),
FinalMatrix (2,2), FinalMatrix(2,3));
  printf ("    | %6.3f %6.3f %6.3f %6.3f | \n", FinalMatrix (3,0), FinalMatrix (3,1),
FinalMatrix (3,2), FinalMatrix(3,3));
  printf ("\n");


boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer2;
viewer= simpleVis(cloud_out, ICPFinalPtr);
viewer2= simpleVis(finalPtr, cloud_in, cloud_out, ICPFinalPtr);


while (!viewer->wasStopped ())
  {
    viewer->spinOnce (100);
    boost::this_thread::sleep (boost::posix_time::microseconds (100000));
  }
while (!viewer2->wasStopped ())
  {
    viewer2->spinOnce (100);
```

```
    boost::this_thread::sleep (boost::posix_time::microseconds (100000));
  }

*cloud_out += ICPFinal;

  pcl::io::savePCDFile("ConcatenatedFields.pcd", *cloud_out);
}
```

Concatenation Code (Part 2)

```
#include <iostream>
#include <Eigen/Dense>
#include <boost/thread/thread.hpp>
#include <pcl/common/common_headers.h>
#include <pcl/features/normal_3d.h>
#include <pcl/io/pcd_io.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/console/parse.h>
#include <pcl/point_types.h>
#include <pcl/registration/icp.h>
#include <limits>
#include <vector>
#include <Eigen/Core>
#include <pcl/point_cloud.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/fpfh.h>
#include <pcl/registration/ia_ransac.h>
#include <pcl/common/transforms.h>
using Eigen::Matrix4f;
boost::shared_ptr<pcl::visualization::PCLVisualizer> simpleVis
(pcl::PointCloud<pcl::PointXYZ>::ConstPtr Cloud1,

                    pcl::PointCloud<pcl::PointXYZ>::ConstPtr Cloud2)



{
       boost::shared_ptr<pcl::visualization::PCLVisualizer> Viewer (new
pcl::visualization::PCLVisualizer ("3D Viewer"));
       Viewer->setBackgroundColor (0, 0, 0);
       pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> Cloud1_Color
(Cloud1, 0, 255, 0);
       pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> Cloud2_Color
(Cloud2, 255, 0, 0);
       //pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ> Cloud3_Color
(Cloud3, 0, 255, 0);
       Viewer->addPointCloud<pcl::PointXYZ> (Cloud1, Cloud1_Color, "Cloud1");
       Viewer->addPointCloud<pcl::PointXYZ> (Cloud2, Cloud2_Color, "Cloud2");
       //Viewer->addPointCloud<pcl::PointXYZ> (Cloud3, Cloud3_Color, "Cloud3");
       Viewer->addCoordinateSystem(1.0);
       return(Viewer);
}


int
```

```
main(int argc, char** argv)
{
pcl::PointCloud<pcl::PointXYZ> Cloud0;
pcl::PointCloud<pcl::PointXYZ> Cloud1;
pcl::PointCloud<pcl::PointXYZ> Cloud1a;
pcl::PointCloud<pcl::PointXYZ> Cloud2;
pcl::PointCloud<pcl::PointXYZ> Cloud2a;
pcl::PointCloud<pcl::PointXYZ> Cloud3;
pcl::PointCloud<pcl::PointXYZ> Cloud3a;
pcl::PointCloud<pcl::PointXYZ> Cloud4;
pcl::PointCloud<pcl::PointXYZ> Cloud4a;
pcl::PointCloud<pcl::PointXYZ> Cloud5;
pcl::PointCloud<pcl::PointXYZ> Cloud5a;
       pcl::io::loadPCDFile("IncrementalDockData\\Location 310cm.pcd", Cloud5);
       pcl::io::loadPCDFile("IncrementalDockData\\Location 610cm.pcd", Cloud4);
       pcl::io::loadPCDFile("IncrementalDockData\\Location 910cm.pcd", Cloud3);
       pcl::io::loadPCDFile("IncrementalDockData\\Location 1210cm.pcd", Cloud2);
       pcl::io::loadPCDFile("IncrementalDockData\\Location 1510cm.pcd", Cloud1);
       pcl::io::loadPCDFile("IncrementalDockData\\Location 1810cm.pcd", Cloud0);

Matrix4f m54(4,4); //Example of how to declare a transformation matrix
m54(0,0)=.997;
m54(1,0)=.074;
m54(2,0)=-.021;
m54(3,0)=0;
m54(0,1)=-.074;
m54(1,1)=.997;
m54(2,1)=-.004;
m54(3,1)=0;
m54(0,2)=.021;
m54(1,2)=.006;
m54(2,2)=1;
m54(3,2)=0;
m54(0,3)=-2.943;
m54(1,3)=1;
m54(2,3)=-.2;
m54(3,3)=1;

Matrix4f m43(4,4); // Have to declare the other matrices here

Matrix4f m32(4,4);

Matrix4f m21(4,4);

Matrix4f m10(4,4);

pcl::transformPointCloud(Cloud5, Cloud5a, m54*m43*m32*m21*m10);
pcl::transformPointCloud(Cloud4, Cloud4a, m43*m32*m21*m10);
pcl::transformPointCloud(Cloud3, Cloud3a, m32*m21*m10);
pcl::transformPointCloud(Cloud2, Cloud2a, m21*m10);
pcl::transformPointCloud(Cloud1, Cloud1a, m10);


Cloud0 +=Cloud1a;
Cloud0 +=Cloud2a;
Cloud0 +=Cloud3a;
Cloud0 +=Cloud4a;
Cloud0 +=Cloud5a;
```

```
pcl::io::savePCDFile("TransformAttempt.pcd", Cloud0);

}
```

Algorithm Code (Part 3)

```
// Segmentation.cpp : Defines the entry point for the console application.
//

#include <iostream>
#include <fstream>
#include <pcl/ModelCoefficients.h>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <boost/thread/thread.hpp>
#include <pcl/common/common_headers.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/console/parse.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/common/centroid.h>
#include <pcl/surface/concave_hull.h>
#include <Eigen/Core>
#include <Eigen/Dense>
#include <Eigen/StdVector>
#include <Eigen/Eigenvalues>
#include <Eigen/SVD>
#include <Eigen/LU>
#include <Eigen/Geometry>
#include <pcl/common/eigen.h>
#include <pcl/filters/passthrough.h>
#include <pcl/common/common.h>
#include <cmath>
#include <math.h>
#include <stdio.h>
#include <cv.h>
#include <core_c.h>
#include <mat.hpp>
#include <core.hpp>
#include <opencv2\core\mat.hpp>
#include <opencv2\core\core.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\highgui\highgui_c.h>
#include <opencv2\gpu\gpumat.hpp>
#include <opencv2\gpu\gpu.hpp>
#include <opencv2\imgproc\imgproc.hpp>

//using namespace cv;
using namespace std;
using Eigen::Matrix2f;

boost::shared_ptr<pcl::visualization::PCLVisualizer>
```

```
simpleVis (pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_in,
pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_out)
// above declares that four things will be present in the viewer: cloud_in,
cloud_normals1, cloud_out, and cloud_normals2.

{


  // --------------------------------------------
  // -----Open 3D viewer and add point cloud-----
  // --------------------------------------------
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new
pcl::visualization::PCLVisualizer ("3D Viewer"));// creates object viewer
  viewer->setBackgroundColor (0, 0, 0);

  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_in_color(cloud_in, 0, 255, 0);// sets cloud_in, which will be cloud out
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_out_color(cloud_out, 255, 0, 0); // sets cloud_out to blue
  viewer->addPointCloud<pcl::PointXYZ> (cloud_in, cloud_in_color, "cloud_in"); //adds
point cloud and defines it as cloud_in
  viewer->addPointCloud<pcl::PointXYZ> (cloud_out, cloud_out_color, "cloud_out"); //adds
point cloud and defines it as cloud_out
  viewer->setPointCloudRenderingProperties
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 2, "cloud_in");
  viewer->setPointCloudRenderingProperties
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 2, "cloud_out");
  viewer->addCoordinateSystem (1.0);
  viewer->initCameraParameters ();
  viewer->camera_.clip[0] = 19.9213;
  viewer->camera_.clip[1] = 323.021;
  viewer->camera_.focal[0] = -38.4332;
  viewer->camera_.focal[1] = -1.29291;
  viewer->camera_.focal[2] = -3.64906;
  viewer->camera_.pos[0] = 94.565;
  viewer->camera_.pos[1] = -23.85;
  viewer->camera_.pos[2] = 71.8934;
  viewer->camera_.view[0] = -0.49417;
  viewer->camera_.view[1] = -0.00221676;
  viewer->camera_.view[2] = 0.869362;
  viewer->camera_.window_size[0] = 1366;
  viewer->camera_.window_size[1] = 706;
  viewer->camera_.window_pos[0] = 0;
  viewer->camera_.window_pos[1] = 0;
  viewer->updateCamera();
  return (viewer);
}
boost::shared_ptr<pcl::visualization::PCLVisualizer>
simpleVis (pcl::PointCloud<pcl::PointXYZ>::ConstPtr cloud_in)


{


  // --------------------------------------------
  // -----Open 3D viewer and add point cloud-----
  // --------------------------------------------
```

```
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer2 (new
pcl::visualization::PCLVisualizer ("3D Viewer"));// creates object viewer
  viewer2->setBackgroundColor (0, 0, 0);

  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_in_color(cloud_in, 0, 255, 0);// sets cloud_in, which will be cloud out
  //pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZ>
cloud_out_color(cloud_out, 255, 255, 0); // sets cloud_out to blue
  viewer2->addPointCloud<pcl::PointXYZ> (cloud_in, cloud_in_color, "cloud_in"); //adds
point cloud and defines it as cloud_in
  //viewer->addPointCloud<pcl::PointXYZ> (cloud_out, cloud_out_color, "cloud_out");
//adds point cloud and defines it as cloud_out
  viewer2->setPointCloudRenderingProperties
(pcl::visualization::PCL_VISUALIZER_POINT_SIZE, 2, "cloud_in");
  viewer2->addCoordinateSystem (1.0);
  viewer2->initCameraParameters ();
  viewer2->camera_.clip[0] = 19.9213;
  viewer2->camera_.clip[1] = 323.021;
  viewer2->camera_.focal[0] = -38.4332;
  viewer2->camera_.focal[1] = -1.29291;
  viewer2->camera_.focal[2] = -3.64906;
  viewer2->camera_.pos[0] = 94.565;
  viewer2->camera_.pos[1] = -23.85;
  viewer2->camera_.pos[2] = 71.8934;
  viewer2->camera_.view[0] = -0.49417;
  viewer2->camera_.view[1] = -0.00221676;
  viewer2->camera_.view[2] = 0.869362;
  viewer2->camera_.window_size[0] = 1366;
  viewer2->camera_.window_size[1] = 706;
  viewer2->camera_.window_pos[0] = 0;
  viewer2->camera_.window_pos[1] = 0;

  /*viewer2->camera_.clip[0] = 275.235;
  viewer2->camera_.clip[1] = 386.96;
  viewer2->camera_.focal[0] = -31.3541;
  viewer2->camera_.focal[1] = 6.93915;
  viewer2->camera_.focal[2] = -3.87708;
  viewer2->camera_.pos[0] = -72.4913;
  viewer2->camera_.pos[1] = 26.9881;
  viewer2->camera_.pos[2] = 326.9881;
  viewer2->camera_.view[0] = 0.0299683;
  viewer2->camera_.view[1] = 0.997936;
  viewer2->camera_.view[2] = -0.0567898;
  viewer2->camera_.window_size[0] = 1366;
  viewer2->camera_.window_size[1] = 706;
  viewer2->camera_.window_pos[0] = 0;
  viewer2->camera_.window_pos[1] = 0;*/
  viewer2->updateCamera();
  return (viewer2);
}


int
main (int argc, char** argv)
{
pcl::PointCloud<pcl::PointXYZ> Cloud;
pcl::PointCloud<pcl::PointXYZ> Cloud7;
pcl::PointCloud<pcl::PointXYZ> Cloud8;
```

```
pcl::PointCloud<pcl::PointXYZ> CloudOutliers;
pcl::PointCloud<pcl::PointXYZ> CloudInliers;
pcl::PointCloud<pcl::PointXYZ> CloudXY;
pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
pcl::PointIndices::Ptr inliers (new pcl::PointIndices);

//Uncomment the following lines to load different point clouds into the algorithm

//pcl::io::loadPCDFile("Location 11 Scan 1 ( Elevated).pcd", Cloud8);
//pcl::io::loadPCDFile("Location 12 Scan 1 (Elevated).pcd", Cloud7);
//pcl::io::loadPCDFile("TransformAttempt310-1810.pcd", Cloud);
pcl::io::loadPCDFile("TransformAttemptLine4-0-900cm.pcd", Cloud);
//pcl::io::loadPCDFile("TransformAttemptDock2.pcd", Cloud);
//pcl::io::loadPCDFile("Transform11-10.pcd", Cloud);
//pcl::io::loadPCDFile("Location 1 Scan 1.pcd", Cloud);
//pcl::io::loadPCDFile("Location 1 Scan 3 (Elevated).pcd", Cloud);

//*******************************************************************************
// This section is the start of the Shore Removal Stage
//*******************************************************************************

//Get cloud dimensions and round value to nearest integer

Eigen::Vector4f min_pt_cloud;
Eigen::Vector4f max_pt_cloud;
pcl::getMinMax3D(Cloud, min_pt_cloud, max_pt_cloud);
signed int Xmax_cloud, Ymax_cloud, Xmin_cloud, Ymin_cloud;
Xmax_cloud = ceil(max_pt_cloud[0]);
Ymax_cloud = ceil(max_pt_cloud[1]);
Xmin_cloud = floor(min_pt_cloud[0]);
Ymin_cloud = floor(min_pt_cloud[1]);

//Veiw dimensions of cloud
std::cout<<"Xmax "<< Xmax_cloud<<"Xmin "<< Xmin_cloud<<"Ymax "<< Ymax_cloud<<"Xmin "<<
Ymin_cloud<<""<<endl;

// Declare what each pixel will represent in size, i.e. size one means pixel represents 1
// by 1 meter square of the point cloud, and a value of 0.5 would represent a 0.5 by 0.5
// box
float Pixelsize = 1;

//Set number of pixels in row and columns in image based on size of point cloud and
// desired resolution
int SizeY;
SizeY = pow(Pixelsize, -1)*((Ymax_cloud-Ymin_cloud)+2);
int SizeX;
SizeX = pow(Pixelsize, -1)*((Xmax_cloud-Xmin_cloud)+2);
//std::cout<<" SizeX  is " << SizeX << " and SizeY is "<<SizeY<<""<<endl;

// Creates empty image
cv::Mat image = cv::Mat::zeros(SizeY, SizeX, CV_8U);

//-------------------------------------------------------------------------------
// This is the code that converts the point cloud to a binary image
//-------------------------------------------------------------------------------

int PixelY;
int PixelX;
```

```
for(signed int y =(Ymin_cloud -1); y != (Ymax_cloud +1); y++)
{
//std::cout<<"  Y value is "<< y<< ""<<endl;
	for(signed int x = (Xmin_cloud -1); x !=(Xmax_cloud+1); x++)
	{
		//std::cout<<"  X value is "<< x << ""<<endl;
		for(size_t z = 0; z < Cloud.points.size (); z++)
		{
			if ( Cloud.points[z].x >= x && Cloud.points[z].x < x+1 &&
Cloud.points[z].y >= y && Cloud.points[z].y < y+1)
			{
				PixelY = (pow(Pixelsize,-1)*y)+(pow(Pixelsize,-
1)*(abs(Ymin_cloud)+1));
				PixelX = ((pow(Pixelsize,-1)*x) + (pow(Pixelsize,-
1)*(abs(Xmin_cloud)+1)));
				//std::cout<<" Pixel Y Value is "<<PixelY<< "and Pixel X value
is "<< PixelX<<""<<endl;

				image.at<uchar>(PixelY, PixelX) = 255;
			}
		}
	}
}
//std::cout<<""<<image<<""<<endl;

// This section declares the rest of the images that will be used in the algorithm for
// the various morphological processes

cv::Mat image2 = cv::Mat::zeros(4*SizeY, 4*SizeX, CV_8U);//Larger grayscale image
cv::Mat image3(4*SizeY, 4*SizeX, CV_8U);//Will be binary image from image2
cv::Mat image3b(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image4(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image5(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image6(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image7(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image8(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image9(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image10(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image11(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image12(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image14(4*SizeY, 4*SizeX, CV_8U);
cv::Mat image15(SizeY, SizeX, CV_8U);
cv::Mat image16(4*SizeY, 4*SizeX, CV_8U);
cv::namedWindow("Binary Image");//Creates window
cv::imshow("Binary Image", image);

// Makes the original image easier to view by flipping the matrix and resizing the image

cv::resize(image, image16, image14.size());
cv::flip(image16, image16,0);
cv::namedWindow("Original Image");
cv::imshow("Original Image", image16);
//cv::imwrite("Original_Image.jpg", image16);

//This section uses contours to eliminate connected pilings below a certain size

CvMemStorage *storage;
CvSeq *Contour = NULL;
```

```
IplImage *input = NULL;
double area;
int FoundContours =0;


IplImage* CopyImage = new IplImage(image);
input = cvCloneImage(CopyImage);
storage = cvCreateMemStorage(0);
CvScalar white, black;
white = CV_RGB(255,255,255);
black = CV_RGB(0, 0, 0);
int size = 20;

cvFindContours(input, storage, &Contour, sizeof(CvContour), CV_RETR_TREE,
CV_CHAIN_APPROX_SIMPLE, cv::Point(0,0));

//for( int i =0; i <contours.size(); i++)
while(Contour)
{
	area = cvContourArea(Contour, CV_WHOLE_SEQ);
	if (0 <= area && area < size)
	{
	cvDrawContours(CopyImage, Contour, black, black, -1, CV_FILLED, 8 );
	}
	else
	{
	if(0<area && area<=size)
	cvDrawContours(CopyImage, Contour, white, white, -1, CV_FILLED, 8 );
	}
	Contour = Contour->h_next;
	FoundContours= FoundContours +1;
}
std::cout<<" Found Contours = "<< FoundContours<<endl;
//return(FoundContours);
cv::Mat image13 (CopyImage);
cv::resize(image13, image14, image14.size());
cv::flip(image14, image14,0);
cv::namedWindow("Contours");
cv::imshow("Contours", image14);
//cv::imwrite("Small_Pixels_Removed.jpg", image14);

int element_size = 4;
cv::resize(image, image2, image2.size());
cv::flip(image2, image2, 0);

// First dilation process

cv::Mat element(2*element_size + 1, 2*element_size+1, CV_8U);
cv::dilate(image2, image3, element, cv::Point(element_size,element_size), 1,
cv::BORDER_CONSTANT );// dilate the image once
cv::namedWindow("Dilated Image");
cv::threshold(image3, image3b, 1, 255, cv::THRESH_BINARY);
cv::imshow("Dilated Image", image3b);
//cv::imwrite("First_Dilated_Image.jpg", image3b);

// Erosion process
```

```
cv::erode(image3b, image4, element, cv::Point(element_size, element_size), 3,
cv::BORDER_CONSTANT );// the 2 erodes the image twice
cv::namedWindow("Eroded Image");
cv::threshold(image4, image5, 1, 255, cv::THRESH_BINARY); // Thresholds image2 into
output image3
cv::imshow("Eroded Image", image5);
//cv::imwrite("Eroded_Image.jpg", image5);
// Second dilation process

cv::dilate(image5, image6, element, cv::Point(element_size, element_size), 2,
cv::BORDER_CONSTANT ); //dilate the image
cvNamedWindow("Dilated Image 2");
cv::threshold(image6, image7, 1, 255, cv::THRESH_BINARY);
cv::imshow("Dilated Image 2", image7);
//cv::imwrite("Second_Dilated_Image.jpg", image7);

// Subtraction of shore from original image

image8 = image2 - image7;
cv::namedWindow("Subtracted Image");
cv::imshow("Subtracted Image", image8);
//cv::imwrite("Subtracted_Image.jpg", image8);

cv::resize(image8, image15, image.size());
cv::flip(image15, image15, 0);

//-------------------------------------------------------------------------------------
// Following section converts binary image of shore removed to a point cloud by
// extracting all the points within its true values
//-------------------------------------------------------------------------------------

int Xminpixel;
int Yminpixel;
pcl::PointCloud<pcl::PointXYZ> ShoreEliminated;
pcl::PointCloud<pcl::PointXYZ>::Ptr ShoreEliminatedPointer (new
pcl::PointCloud<pcl::PointXYZ>);
for(signed int y =0; y != image15.rows; y++)
{
//std::cout<<"  Y value is "<< y<< ""<<endl;
       for(signed int x = 0; x !=image15.cols; x++)
       {
              if(image15.at<uchar>(y,x) > 0)
              {
                     Xminpixel = (x -(pow(Pixelsize,-1)*((abs(Xmin_cloud)
+1)))/pow(Pixelsize,-1));
                     Yminpixel = (y -(pow(Pixelsize,-1)*((abs(Ymin_cloud)
+1)))/pow(Pixelsize,-1));
                     //std::cout<<"XminPixel "<< Xminpixel<< " Yminpixel
"<<Yminpixel<<endl;
                     //std::cout<<"  X value is "<< x << ""<<endl;
                     for(size_t z = 0; z < Cloud.points.size (); ++z)
                     {

                            if ( Cloud.points[z].x >= Xminpixel && Cloud.points[z].x <
Xminpixel+1 && Cloud.points[z].y >= Yminpixel && Cloud.points[z].y < Yminpixel+1)
                            {
                                   ShoreEliminatedPointer-
>points.push_back(Cloud.points[z]);
```

```
                        }
                    }
                }
        }
}
ShoreEliminated += *ShoreEliminatedPointer;
//std::cout<<"image 3"<< image3<< ""<<endl;
//std::cout<<" SizeX  is " << SizeX << " and SizeY is "<<SizeY<<""<<endl;
//std::cout<<" Pixel Y Value is "<<PixelY<< "and Pixel X value is "<< PixelX<<""<<endl;

//*********************************************************************************************
// Following section is the removal of the plane
//*********************************************************************************************


// Finds the plane using RANSAC

pcl::SACSegmentation<pcl::PointXYZ> seg;
seg.setOptimizeCoefficients(false);
seg.setModelType (pcl::SACMODEL_PLANE);
seg.setMethodType (pcl::SAC_RANSAC);
seg.setMaxIterations(1000);
seg.setDistanceThreshold (.26);
seg.setInputCloud(ShoreEliminated.makeShared());
seg.segment(*inliers, *coefficients);

pcl::ExtractIndices<pcl::PointXYZ> extract (true);

extract.setInputCloud(ShoreEliminated.makeShared());
extract.setIndices (inliers);
extract.setNegative(false);
extract.filter (CloudInliers);
pcl::io::savePCDFile("Plane Inliers1.pcd", CloudInliers);

ofstream textfile2;
textfile2.open ("Cloud1.txt");
for(size_t i=0; i < Cloud.points.size (); ++i)
{
        textfile2<<""<<Cloud.points[i].x<<","<<Cloud.points[i].y<<"\n"<<endl;
}

//textfile2.close();

std::cout<<"Plane contains "<< CloudInliers.height * CloudInliers.width<< " data
points"<<std::endl;

// Extract the points that were not included in the plane

extract.setInputCloud(ShoreEliminated.makeShared());
extract.setIndices (inliers);
extract.setNegative(true);
extract.filter (CloudOutliers);
pcl::io::savePCDFile("Plane Outliers1.pcd", CloudOutliers);

std::cout<<"Outlier point cloud contains "<< CloudOutliers.height * CloudOutliers.width<<
" data points"<<std::endl;
// Used to find lines
```

```
std::cerr << "Output: " << Cloud.makeShared()->width * Cloud.makeShared()->height <<"
Data Points"<< std::endl;
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::PointXYZ>);
tree->setInputCloud(CloudOutliers.makeShared());



//*******************************************************************************
// Starts the piling recognition phase
//*******************************************************************************

// Below is the code for Euclidean Cluster Recognition


std::vector<pcl::PointIndices> cluster_inliers;

pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance(.3);
ec.setMinClusterSize(5);
ec.setMaxClusterSize(200);
ec.setSearchMethod(tree);
ec.setInputCloud(CloudOutliers.makeShared());
ec.extract(cluster_inliers);

pcl::PCDWriter writer;


ofstream textfile; //initiate textfile
//textfile.open ("ClusterData3.txt"); //open textfile

// Decalres Covariance Matrix for the trained class of pilings and not pilings

pcl::PointCloud<pcl::PointXYZ> cluster_cloud1;
pcl::PointCloud<pcl::PointXYZ> Pilings;
Eigen::Matrix<float, 7, 7> CovarianceMatrixPilings;
CovarianceMatrixPilings(0,0) = 0.0583562492492493;
CovarianceMatrixPilings(1,0) = -0.0190657348095124;
CovarianceMatrixPilings(2,0) = -0.00224956680661268;
CovarianceMatrixPilings(3,0) = -0.00535996028010200;
CovarianceMatrixPilings(4,0) = 0.000246012726655487;
CovarianceMatrixPilings(5,0) = -0.0374002175288239;
CovarianceMatrixPilings(6,0) = -0.000642917495801380;
CovarianceMatrixPilings(0,1) = -0.0190657348095124;
CovarianceMatrixPilings(1,1) = 0.0462733803806834;
CovarianceMatrixPilings(2,1) = 0.00459550049951883;
CovarianceMatrixPilings(3,1) = 0.00245484674934939;
CovarianceMatrixPilings(4,1) = 0.00490618148773976;
CovarianceMatrixPilings(5,1) = 0.0476752121153300;
CovarianceMatrixPilings(6,1) = 0.0562032359241755;
CovarianceMatrixPilings(0,2) = -0.00224956680661268;
CovarianceMatrixPilings(1,2) = 0.00459550049951883;
CovarianceMatrixPilings(2,2) = 0.0148394399958443;
CovarianceMatrixPilings(3,2) = 0.0143981502225808;
CovarianceMatrixPilings(4,2) = 0.0102616807670333;
CovarianceMatrixPilings(5,2) = -0.0505770456513280;
CovarianceMatrixPilings(6,2) = -0.0755976583582640;
CovarianceMatrixPilings(0,3) = -0.00535996028010168;
CovarianceMatrixPilings(1,3) = -0.00245484674934939;
```

```
CovarianceMatrixPilings(2,3) = 0.0143981502225810;
CovarianceMatrixPilings(3,3) = 0.0282440133402064;
CovarianceMatrixPilings(4,3) = -8.16450786758868e-05;
CovarianceMatrixPilings(5,3) = 0.000375019873311427;
CovarianceMatrixPilings(6,3) = -0.114415792459341;
CovarianceMatrixPilings(0,4) = 0.000246012726655487;
CovarianceMatrixPilings(1,4) = 0.00490618148773976;
CovarianceMatrixPilings(2,4) = 0.0102616807670333;
CovarianceMatrixPilings(3,4) = -8.16450786758868e-05;
CovarianceMatrixPilings(4,4) = 0.0148495108787500;
CovarianceMatrixPilings(5,4) = -0.0739541867751970;
CovarianceMatrixPilings(6,4) = -0.0260239782801664;
CovarianceMatrixPilings(0,5) = -0.0374002175288239;
CovarianceMatrixPilings(1,5) = 0.0476752121153300;
CovarianceMatrixPilings(2,5) = -0.0505770456513280;
CovarianceMatrixPilings(3,5) = 0.000375019873311427;
CovarianceMatrixPilings(4,5) = -0.0739541867751967;
CovarianceMatrixPilings(5,5) = 0.520320249730991;
CovarianceMatrixPilings(6,5) = 0.266422802577038;
CovarianceMatrixPilings(0,6) = -0.000642917495801380;
CovarianceMatrixPilings(1,6) = 0.0562032359241755;
CovarianceMatrixPilings(2,6) = -0.0755976583582640;
CovarianceMatrixPilings(3,6) = -0.114415792459341;
CovarianceMatrixPilings(4,6) = -0.0260239782801664;
CovarianceMatrixPilings(5,6) = 0.266422802577038;
CovarianceMatrixPilings(6,6) = 0.669959376077710;
//std::cout<< " " << CovarianceMatrixPilings<< "\n "<<endl;

Eigen::Matrix<float, 7, 7> InverseCovarianceMatrixPilings;
InverseCovarianceMatrixPilings(0,0) = 23.2061319422865;
InverseCovarianceMatrixPilings(1,0) = 13.4944174578186;
InverseCovarianceMatrixPilings(2,0) = -2.03269798790532;
InverseCovarianceMatrixPilings(3,0) = -37.6312310658365;
InverseCovarianceMatrixPilings(4,0) = 20.1738170017152;
InverseCovarianceMatrixPilings(5,0) = 8.41761247806922;
InverseCovarianceMatrixPilings(6,0) = -10.3296196911351;
InverseCovarianceMatrixPilings(0,1) = 13.4944174578182;
InverseCovarianceMatrixPilings(1,1) = 153.739548919322;
InverseCovarianceMatrixPilings(2,1) = 198.439467130184;
InverseCovarianceMatrixPilings(3,1) = -309.761705276175;
InverseCovarianceMatrixPilings(4,1) = -408.080467423113;
InverseCovarianceMatrixPilings(5,1) = -26.7083456988526;
InverseCovarianceMatrixPilings(6,1) = -48.6241738741668;
InverseCovarianceMatrixPilings(0,2) = -2.03269798792185;
InverseCovarianceMatrixPilings(1,2) = 198.439467130085;
InverseCovarianceMatrixPilings(2,2) = 3205.04482449267;
InverseCovarianceMatrixPilings(3,2) = -2060.88025143045;
InverseCovarianceMatrixPilings(4,2) = -2549.47936741061;
InverseCovarianceMatrixPilings(5,2) = -16.8209401878553;
InverseCovarianceMatrixPilings(6,2) = -99.2951610843068;
InverseCovarianceMatrixPilings(0,3) = -37.6312310658300;
InverseCovarianceMatrixPilings(1,3) = -309.761705276148;
InverseCovarianceMatrixPilings(2,3) = -2060.88025143097;
InverseCovarianceMatrixPilings(3,3) = 2063.19700169003;
InverseCovarianceMatrixPilings(4,3) = 1622.04826303588;
InverseCovarianceMatrixPilings(5,3) = -65.9025771793146;
InverseCovarianceMatrixPilings(6,3) = 234.969639809516;
InverseCovarianceMatrixPilings(0,4) = 20.1738170017294;
```

```
InverseCovarianceMatrixPilings(1,4) = -408.080467423025;
InverseCovarianceMatrixPilings(2,4) = -2549.47936741051;
InverseCovarianceMatrixPilings(3,4) = 1622.04826303539;
InverseCovarianceMatrixPilings(4,4) = 2982.98849474081;
InverseCovarianceMatrixPilings(5,4) = 178.839852850429;
InverseCovarianceMatrixPilings(6,4) = 68.3383123262888;
InverseCovarianceMatrixPilings(0,5) = 8.41761247806985;
InverseCovarianceMatrixPilings(1,5) = -26.7083456988467;
InverseCovarianceMatrixPilings(2,5) = -16.8209401877854;
InverseCovarianceMatrixPilings(3,5) = -65.9025771793640;
InverseCovarianceMatrixPilings(4,5) = 178.839852850372;
InverseCovarianceMatrixPilings(5,5) = 38.7149868350255;
InverseCovarianceMatrixPilings(6,5) = -19.3531776835518;
InverseCovarianceMatrixPilings(0,6) = -10.3296196911355;
InverseCovarianceMatrixPilings(1,6) = -48.6241738741724;
InverseCovarianceMatrixPilings(2,6) = -99.2951610844277;
InverseCovarianceMatrixPilings(3,6) = 234.969639809578;
InverseCovarianceMatrixPilings(4,6) = 68.3383123263913;
InverseCovarianceMatrixPilings(5,6) = -19.3531776835486;
InverseCovarianceMatrixPilings(6,6) = 44.8363204793429;

Eigen::Matrix<float, 1, 7> MeanPilings;
MeanPilings(0,0) = -3.36512702702703;
MeanPilings(0,1) = 1.60986215883237;
MeanPilings(0,2) = 0.398576687408703;
MeanPilings(0,3) = 0.720446584049003;
MeanPilings(0,4) = 0.553345847825569;
MeanPilings(0,5) = 3.03936062845996;
MeanPilings(0,6) = 3.04247673542710;

Eigen::Matrix<float, 7, 7> CovarianceMatrixNotPilings; // Decalres Covariance Matrix for
the trained class of pilings
CovarianceMatrixNotPilings(0,0) = 5.83387605811068;
CovarianceMatrixNotPilings(1,0) = 1.07280369758292;
CovarianceMatrixNotPilings(2,0) = -0.213675715163691;
CovarianceMatrixNotPilings(3,0) = -0.717629117386471;
CovarianceMatrixNotPilings(4,0) = 0.0197402706516416;
CovarianceMatrixNotPilings(5,0) = 2.03280691502565;
CovarianceMatrixNotPilings(6,0) = 1.69758506659330;
CovarianceMatrixNotPilings(0,1) = 1.07280369758292;
CovarianceMatrixNotPilings(1,1) = 4.05729301065676;
CovarianceMatrixNotPilings(2,1) = 0.0869132879462066;
CovarianceMatrixNotPilings(3,1) = -0.00117156543344151;
CovarianceMatrixNotPilings(4,1) = 0.0456354114933153;
CovarianceMatrixNotPilings(5,1) = 8.83354142155646;
CovarianceMatrixNotPilings(6,1) = 5.33983278786738;
CovarianceMatrixNotPilings(0,2) = -0.213675715163691;
CovarianceMatrixNotPilings(1,2) = 0.0869132879462066;
CovarianceMatrixNotPilings(2,2) = 0.280809083796171;
CovarianceMatrixNotPilings(3,2) = 0.502257020522342;
CovarianceMatrixNotPilings(4,2) = 0.0348353797069033;
CovarianceMatrixNotPilings(5,2) = -0.146846515825577;
CovarianceMatrixNotPilings(6,2) = -0.221454553877868;
CovarianceMatrixNotPilings(0,3) = -0.717629117386471;
CovarianceMatrixNotPilings(1,3) = -0.00117156543344151;
CovarianceMatrixNotPilings(2,3) = 0.502257020522342;
CovarianceMatrixNotPilings(3,3) = 1.62365364335768;
CovarianceMatrixNotPilings(4,3) = -0.0698988962946746;
```

```
CovarianceMatrixNotPilings(5,3) = 0.724027180240994;
CovarianceMatrixNotPilings(6,3) = -0.948715461649621;
CovarianceMatrixNotPilings(0,4) = 0.0197402706516416;
CovarianceMatrixNotPilings(1,4) = 0.0456354114933153;
CovarianceMatrixNotPilings(2,4) = 0.0348353797069033;
CovarianceMatrixNotPilings(3,4) = -0.0698988962946746;
CovarianceMatrixNotPilings(4,4) = 0.0442685829367980;
CovarianceMatrixNotPilings(5,4) = -0.282989171174374;
CovarianceMatrixNotPilings(6,4) = 0.111300834494996;
CovarianceMatrixNotPilings(0,5) = 2.03280691502565;
CovarianceMatrixNotPilings(1,5) = 8.83354142155646;
CovarianceMatrixNotPilings(2,5) = -0.146846515825577;
CovarianceMatrixNotPilings(3,5) = 0.724027180240994;
CovarianceMatrixNotPilings(4,5) = -0.282989171174374;
CovarianceMatrixNotPilings(5,5) = 26.1648589365022;
CovarianceMatrixNotPilings(6,5) = 10.8986425007864;
CovarianceMatrixNotPilings(0,6) = 1.69758506659330;
CovarianceMatrixNotPilings(1,6) = 5.33983278786738;
CovarianceMatrixNotPilings(2,6) = -0.221454553877868;
CovarianceMatrixNotPilings(3,6) = -0.948715461649621;
CovarianceMatrixNotPilings(4,6) = 0.111300834494996;
CovarianceMatrixNotPilings(5,6) = 10.8986425007864;
CovarianceMatrixNotPilings(6,6) = 8.29172075686866;

Eigen::Matrix<float, 7, 7> InverseCovarianceMatrixNotPilings;
InverseCovarianceMatrixNotPilings(0,0) = 0.192914571433160;
InverseCovarianceMatrixNotPilings(1,0) = -0.129414556216729;
InverseCovarianceMatrixNotPilings(2,0) = -0.00174220149071247;
InverseCovarianceMatrixNotPilings(3,0) = 0.111422779242463;
InverseCovarianceMatrixNotPilings(4,0) = 0.205934806603797;
InverseCovarianceMatrixNotPilings(5,0) = 0.0120106529682960;
InverseCovarianceMatrixNotPilings(6,0) = 0.0379975846296558;
InverseCovarianceMatrixNotPilings(0,1) = -0.129414556216729;
InverseCovarianceMatrixNotPilings(1,1) = 5.66348898355880;
InverseCovarianceMatrixNotPilings(2,1) = -4.62721055750378;
InverseCovarianceMatrixNotPilings(3,1) = 0.257879384064185;
InverseCovarianceMatrixNotPilings(4,1) = -1.14143301080886;
InverseCovarianceMatrixNotPilings(5,1) = -0.898252968699391;
InverseCovarianceMatrixNotPilings(6,1) = -2.51885885036097;
InverseCovarianceMatrixNotPilings(0,2) = -0.00174220149071218;
InverseCovarianceMatrixNotPilings(1,2) = -4.62721055750379;
InverseCovarianceMatrixNotPilings(2,2) = 24.6363672689037;
InverseCovarianceMatrixNotPilings(3,2) = -7.86344432113909;
InverseCovarianceMatrixNotPilings(4,2) = -28.0734474205185;
InverseCovarianceMatrixNotPilings(5,2) = 0.700319791958438;
InverseCovarianceMatrixNotPilings(6,2) = 2.19486597654395;
InverseCovarianceMatrixNotPilings(0,3) = 0.111422779242463;
InverseCovarianceMatrixNotPilings(1,3) = 0.257879384064188;
InverseCovarianceMatrixNotPilings(2,3) = -7.86344432113909;
InverseCovarianceMatrixNotPilings(3,3) = 3.69967809332743;
InverseCovarianceMatrixNotPilings(4,3) = 10.1582517977127;
InverseCovarianceMatrixNotPilings(5,3) = -0.189456004912508;
InverseCovarianceMatrixNotPilings(6,3) = 0.137071087910785;
InverseCovarianceMatrixNotPilings(0,4) = 0.205934806603797;
InverseCovarianceMatrixNotPilings(1,4) = -1.14143301080887;
InverseCovarianceMatrixNotPilings(2,4) = -28.0734474205185;
InverseCovarianceMatrixNotPilings(3,4) = 10.1582517977127;
InverseCovarianceMatrixNotPilings(4,4) = 77.5620551575260;
```

```
InverseCovarianceMatrixNotPilings(5,4) = 1.64155809610995;
InverseCovarianceMatrixNotPilings(6,4) = -2.09337837989663;
InverseCovarianceMatrixNotPilings(0,5) = 0.0120106529682960;
InverseCovarianceMatrixNotPilings(1,5) = -0.898252968699392;
InverseCovarianceMatrixNotPilings(2,5) = 0.700319791958438;
InverseCovarianceMatrixNotPilings(3,5) = -0.189456004912507;
InverseCovarianceMatrixNotPilings(4,5) = 1.64155809610995;
InverseCovarianceMatrixNotPilings(5,5) = 0.304881153981280;
InverseCovarianceMatrixNotPilings(6,5) = 0.150268353575219;
InverseCovarianceMatrixNotPilings(0,6) = 0.0379975846296558;
InverseCovarianceMatrixNotPilings(1,6) = -2.51885885036097;
InverseCovarianceMatrixNotPilings(2,6) = 2.19486597654395;
InverseCovarianceMatrixNotPilings(3,6) = 0.137071087910785;
InverseCovarianceMatrixNotPilings(4,6) = -2.09337837989664;
InverseCovarianceMatrixNotPilings(5,6) = 0.150268353575218;
InverseCovarianceMatrixNotPilings(6,6) = 1.63984769452969;

Eigen::Matrix<float, 1, 7> MeanNotPilings;
MeanNotPilings(0,0) = -1.84469046494845;
MeanNotPilings(0,1) = 1.50911383574009;
MeanNotPilings(0,2) = 0.576742731909785;
MeanNotPilings(0,3) = 1.63000391704224;
MeanNotPilings(0,4) = 0.396269612773835;
MeanNotPilings(0,5) = 4.51507139357617;
MeanNotPilings(0,6) = 1.89252992888128;

// The following code iterates through each cluster, adds in the data points from the
//planar surface, and then uses a Gaussian PDF and Bayes' Theorem to determine if each
//cluster is a piling

double DeterminantCovarianceMatrixPilings;
double DeterminantCovarianceMatrixNotPilings;

DeterminantCovarianceMatrixPilings = 2.362248032597543e-013;
DeterminantCovarianceMatrixNotPilings = 0.140828758281363;

int n =0;

Eigen::MatrixXf InlierCentroids(200,3);
int j=0;
const double Pi = 4.0*atan(1.0);
for (std::vector<pcl::PointIndices>::const_iterator it = cluster_inliers.begin (); it
!=cluster_inliers.end (); ++it)
{
	pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new
pcl::PointCloud<pcl::PointXYZ>);
	for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it-
>indices.end (); pit++)
		cloud_cluster->points.push_back(CloudOutliers.makeShared()->points[*pit]);
	cloud_cluster->width= cloud_cluster->points.size ();
	cloud_cluster->height = 1;
	//Declares objects need to get Min and Max values for each cluster
	Eigen::Vector4f min_pt;
	Eigen::Vector4f max_pt;
	pcl::PointCloud<pcl::PointXYZ> cloud_in_unfiltered1;
	pcl::PointCloud<pcl::PointXYZ> cloud_in;
```

```
	// Gets Min and Max for each cluster and then uses those boundaries as a
passthrough filter
	// threshold for the plane inlier cloud.  Then adds all points within the X-Y
boounds of the cluster from
	//the inlier plane to the clusters.
	pcl::getMinMax3D(*cloud_cluster, min_pt, max_pt);
	pcl::PassThrough<pcl::PointXYZ> pass;
	pass.setInputCloud (CloudInliers.makeShared());
	pass.setFilterFieldName ("x");
	pass.setFilterLimits (min_pt[0]-.02, max_pt[0]+.02);
	pass.filter(cloud_in_unfiltered1);
	pass.setInputCloud(cloud_in_unfiltered1.makeShared());
	pass.setFilterFieldName ("y");
	pass.setFilterLimits (min_pt[1]-.02, max_pt[1]+.02);
	pass.filter(cloud_in);
	//CloudInliers.points.erase(cloud_in);
	*cloud_cluster += cloud_in;

	//std::cout << "PointCloud representing the Cluster"<< j <<":  " << cloud_cluster-
>points.size () << " datapoints." << std::endl;
	//Use above line to see data size of each cluster

// Compute Centrod and covariance matrix for each cluster

	Eigen::Matrix <float, 4, 1> CentroidVector;
	pcl::compute3DCentroid(*cloud_cluster,  CentroidVector);// Computes the centroid
for each cluster since command is embedded in the for loop
	//std::cout <<"Centroid for Cluster"<< j <<" is at:\n X:"<< CentroidVector[0] <<
"\n Y:"<< CentroidVector[1]<< "\n Z:" <<CentroidVector[2]<<" " << std::endl;
	Eigen::Matrix3f CovarianceMatrix;
	pcl::computeCovarianceMatrixNormalized(*cloud_cluster, CentroidVector,
CovarianceMatrix);

/*	printf ("\n");
	printf ("    | %10.9f %10.9f %10.9f | \n", CovarianceMatrix (0,0),
CovarianceMatrix (0,1), CovarianceMatrix (0,2));
	printf ("R = | %10.9f %10.9f %10.9f | \n", CovarianceMatrix (1,0),
CovarianceMatrix (1,1), CovarianceMatrix (1,2));
	printf ("    | %10.9f %10.9f %10.9f | \n", CovarianceMatrix (2,0),
CovarianceMatrix (2,1), CovarianceMatrix (2,2));
	printf ("\n");*/

//Below are the calculations for the seven features for the Gaussian PDF

	double ZNstd;
	double EigShort;
	double EigLong;
	double EigRatio;
	double ZRatioEigRatio;
	double ZRatioEigMean;
	int N=3;


	Eigen::Vector2f Eigenvalues;
	Eigen::Matrix2f Eigenmatrix;
	Eigen::Matrix2f XYCovarianceMatrix = CovarianceMatrix.block<2,2>(0,0);

	pcl::eigen22(XYCovarianceMatrix,Eigenmatrix, Eigenvalues);
```

```
	ZNstd = 2*N*sqrt(CovarianceMatrix(2,2));
	EigShort = 2*N*sqrt(Eigenvalues(0));
	EigLong = 2*N*sqrt(Eigenvalues(1));
	EigRatio = EigShort/EigLong;
	ZRatioEigRatio = ZNstd/EigRatio;
	ZRatioEigMean = ZNstd/((EigShort + EigLong)/2);

	Eigen::Matrix<float, 1, 7> Testdata;
	Testdata(0,0) = CentroidVector[2];
	Testdata(0,1) = ZNstd;
	Testdata(0,2) = EigShort;
	Testdata(0,3) = EigLong;
	Testdata(0,4) = EigRatio;
	Testdata(0,5) = ZRatioEigRatio;
	Testdata(0,6) = ZRatioEigMean;

	//Use to print Testdata, Accurate compared to Matlab
	//std::cout<<" "<< Testdata(0,0)<<" "<< Testdata(0,1)<<" "<<Testdata(0,2)<<"
"<<Testdata(0,3)<<" "<<Testdata(0,4)<<" "<<Testdata(0,5)<<" "<<Testdata(0,6)<<"\n"<<endl;

	//*************************************************************************
*********************************************
	//This section calculates the PDF values for pilings and not pilings of a given
data set
	double PDFPilings;
	Eigen::Matrix<float, 1, 7> PDFPilingsVariable1;
	PDFPilingsVariable1 = (Testdata - MeanPilings) ;
	Eigen::Matrix<float, 7, 1> PDFPilingsVariable2;
	PDFPilingsVariable2 = PDFPilingsVariable1.transpose();
	Eigen::Matrix<float, 1, 7> PDFPilingsVariable3(1,7);
	PDFPilingsVariable3 = PDFPilingsVariable1 * InverseCovarianceMatrixPilings;
	Eigen::Matrix<float, 1, 1> PDFPilingsVariable4;
	PDFPilingsVariable4 = PDFPilingsVariable3* PDFPilingsVariable2;
	float PDFPilingsVariableFinal;
	PDFPilingsVariableFinal = PDFPilingsVariable4(0,0);

	PDFPilings = (1/((pow((Pi*2.0),(7.0/2.0))) *
pow(DeterminantCovarianceMatrixPilings, 0.5)))*(exp(-0.5*(PDFPilingsVariableFinal)));
	//std::cout<< "PDFVariable " << PDFPilings << " " <<std::endl; //Use this to print
PDFPilings

	double PDFNotPilings;
	Eigen::Matrix<float, 1, 7> PDFNotPilingsVariable1;
	PDFNotPilingsVariable1 = (Testdata - MeanNotPilings) ;
	Eigen::Matrix<float, 7, 1> PDFNotPilingsVariable2;
	PDFNotPilingsVariable2 = PDFNotPilingsVariable1.transpose();
	Eigen::Matrix<float, 1, 7> PDFNotPilingsVariable3(1,7);
	PDFNotPilingsVariable3 = PDFNotPilingsVariable1 *
InverseCovarianceMatrixNotPilings;
	Eigen::Matrix<float, 1, 1> PDFNotPilingsVariable4;
	PDFNotPilingsVariable4 = PDFNotPilingsVariable3* PDFNotPilingsVariable2;
	float PDFNotPilingsVariableFinal;
	PDFNotPilingsVariableFinal = PDFNotPilingsVariable4(0,0);

	PDFNotPilings = (1/((pow((Pi*2.0),(7.0/2.0))) *
pow(DeterminantCovarianceMatrixNotPilings, 0.5)))*(exp(-
0.5*(PDFNotPilingsVariableFinal)));
```

```
	//std::cout<< "PDFNotPilingsVariable " << PDFNotPilings << "  " <<std::endl;//Use
this to print PDFNotPilings
	//*************************************************************************
**************************************************
	//std::cout<<"EigenValues are " <<Eigenvalues[0]<< ",
"<<Eigenvalues[1]<<std::endl;
	//std::cout<< " ZNstd is "<< ZNstd <<", EigX is "<< EigX<< ", EigY is "<< EigY <<
", Ratio is " << EigX/EigY<<" for cloud " << j <<std::endl;

	/*textfile <<""<<
CentroidVector[0]<<","<<CentroidVector[1]<<","<<CentroidVector[2]<<"," //writes each
cluster data to a text file

	<<CovarianceMatrix(0,0)<<","<<CovarianceMatrix(0,1)<<","<<CovarianceMatrix(0,2)<<"
,"

	<<CovarianceMatrix(1,0)<<","<<CovarianceMatrix(1,1)<<","<<CovarianceMatrix(1,2)<<"
,"

	<<CovarianceMatrix(2,0)<<","<<CovarianceMatrix(2,1)<<","<<CovarianceMatrix(2,2)<<"
\n"<<endl;*/

	//std:: stringstream ss;//Uncomment next three lines to save each individual
cluster
	//ss<< "cloud_cluster_" << j << ".pcd";
	//writer.write<pcl::PointXYZ> (ss.str(), *cloud_cluster, false);
	//*************************************************************************
**************************************************
	//CALCULATE PROBABILITY OF BEING A PILING
	double ProbabilityX;
	double ProbabilityX1;
	double ProbabilityX2;
	double ProbabilityPilings;
	double ProbabilityNotPilings;
	double Probability1;
	double Probability2;
	ProbabilityPilings= .16296296296;
	ProbabilityNotPilings =.83703703704;
	ProbabilityX1 = ProbabilityPilings*PDFPilings;
	//std::cout<< "ProbabilityX1 =  " << ProbabilityX1 << "  " <<std::endl;
	ProbabilityX2 =  ProbabilityNotPilings*PDFNotPilings;
	//std::cout<< "ProbabilityX2 =  " << ProbabilityX2 << "  " <<std::endl;
	ProbabilityX = ProbabilityX1 + ProbabilityX2;
	//std::cout<< "ProbabilityX =  " << ProbabilityX << "  " <<std::endl;



	Probability1 = (ProbabilityPilings*PDFPilings);
	Probability2 = Probability1/ProbabilityX;
	//std::cout<< ""<<Probability1<<" / "  << ProbabilityX <<" =
"<<Probability2<<""<<endl;

	// Below is the thresholding based off probability as an output of Bayes' Theorem
	if(Probability2 > .5)
	{

		InlierCentroids.row(n) << CentroidVector(0), CentroidVector(1),
CentroidVector(2);
```

```
                Pilings += *cloud_cluster;
                n = n+1;

        }
        cluster_cloud1 += *cloud_cluster;
        j++;
}

//*****************************************************************************************
// Below is the code that compares identified pilings with objects removed from shore
//*****************************************************************************************

//Convert piling point cloud to binary image

cv::Mat PilingImage = cv::Mat::zeros(SizeY, SizeX, CV_8U);;
for(signed int y =(Ymin_cloud -1); y != (Ymax_cloud +1); y++)
{
//std::cout<<"  Y value is "<< y<< ""<<endl;
        for(signed int x = (Xmin_cloud -1); x !=(Xmax_cloud+1); x++)
        {
                //std::cout<<"  X value is "<< x << ""<<endl;
                for(size_t z = 0; z !=n ; z++)
                {
                        if ( InlierCentroids(z,0) >= x && InlierCentroids(z,0) < x+1 &&
InlierCentroids(z,1) >= y && InlierCentroids(z,1) < y+1)
                        {
                                PixelY = (pow(Pixelsize,-1)*y)+(pow(Pixelsize,-
1)*(abs(Ymin_cloud)+1));
                                PixelX = ((pow(Pixelsize,-1)*x) + (pow(Pixelsize,-
1)*(abs(Xmin_cloud)+1)));
                                //std::cout<<" Pixel Y Value is "<<PixelY<< "and Pixel X value
is "<< PixelX<<""<<endl;
                                PixelGrid(PixelY, PixelX) =1;
                                PilingImage.at<uchar>(PixelY, PixelX) = 255;
                        }

                }
        }
}
cv::Mat PilingImage2 = cv::Mat::zeros(4*SizeY, 4*SizeX, CV_8U);
cv::resize(PilingImage, PilingImage2, PilingImage2.size());
cv::flip(PilingImage2, PilingImage2, 0);
cv::namedWindow("Piling Image");
cv::imshow("Piling Image", PilingImage2);
//cv::imwrite("Piling_Image.jpg", PilingImage2);

// Erosion and dilation process

int element_size2 = 7;
int element_size3 = 3;
cv::Mat element2(2*element_size2 + 1, 2*element_size2 + 1, CV_8U);
cv::Mat element3(2*element_size3 + 1, 2*element_size3 + 1, CV_8U);
cv::dilate(PilingImage2, image10, element2, cv::Point(element_size2,element_size2), 1,
cv::BORDER_CONSTANT );
cv::erode(image10, image10, element3, cv::Point(element_size3,element_size3), 1,
cv::BORDER_CONSTANT );
cv::threshold(image10, image10, 1, 255, cv::THRESH_BINARY);
```

```
cv::namedWindow("Dilated Pilings");
cv::imshow("Dilated Pilings", image10);
//cv::imwrite("Dilated_Pilings.jpg", image10);

// Multiply pilings by objects removed from shore to verify that both are true

cv::multiply(image8 , image10, image9, 1);
cv::namedWindow("Pilings Multiplied with Morphology");
cv::imshow("Pilings Multiplied with Morphology", image9);

cv::resize(image9, image11, image.size());
cv::flip(image11, image11, 0);
cv::namedWindow("Dock");
cv::imshow("Dock", image11);

int Xminpixel2;
//int Xmaxpixel;
int Yminpixel2;
//int Ymaxpixel;
pcl::PointCloud<pcl::PointXYZ> IdentifiedDock;
pcl::PointCloud<pcl::PointXYZ>::Ptr IdentifiedDockPointer (new
pcl::PointCloud<pcl::PointXYZ>);
pcl::PointXYZ Position;
size_t counter = 1;

// Convert binary image to point cloud based on binary image's true pixel values

for(signed int y =0; y != image11.rows; y++)
{
//std::cout<<"  Y value is "<< y<< ""<<endl;
       for(signed int x = 0; x !=image11.cols; x++)
       {
              if(image11.at<uchar>(y,x) > 0)
              {
                     Xminpixel2 = (x -(pow(Pixelsize,-1)*((abs(Xmin_cloud)
+1)))/pow(Pixelsize,-1));
                     Yminpixel2= (y -(pow(Pixelsize,-1)*((abs(Ymin_cloud)
+1)))/pow(Pixelsize,-1));
                     //std::cout<<"XminPixel "<< Xminpixel<< " Yminpixel
"<<Yminpixel<<endl;
                     //std::cout<<"  X value is "<< x << ""<<endl;
                     for(size_t z = 0; z < Cloud.points.size (); ++z)
                     {

                            if ( Cloud.points[z].x >= Xminpixel2 && Cloud.points[z].x <
Xminpixel2+1 && Cloud.points[z].y >= Yminpixel2 && Cloud.points[z].y < Yminpixel2+1)
                            {
                                   IdentifiedDockPointer-
>points.push_back(Cloud.points[z]);
                            }
                     }
              }
       }
}
IdentifiedDock += *IdentifiedDockPointer;

std::vector<pcl::PointIndices> cluster_inliers2;
```

```
// Cluster objects from output point cloud

ec.setClusterTolerance(1);
ec.setMinClusterSize(5);
ec.setMaxClusterSize(100000);
ec.setSearchMethod(tree);
ec.setInputCloud(IdentifiedDock.makeShared());
ec.extract(cluster_inliers2);

//Determine the eigenvalues for each cluster and threshold accordingly

pcl::PointCloud<pcl::PointXYZ> IdentifiedDockFinal;

for (std::vector<pcl::PointIndices>::const_iterator it = cluster_inliers2.begin (); it
!=cluster_inliers2.end (); ++it)
{
	pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster2 (new
pcl::PointCloud<pcl::PointXYZ>);
	for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it-
>indices.end (); pit++)
		cloud_cluster2->points.push_back(IdentifiedDock.makeShared()-
>points[*pit]);
	{

		Eigen::Matrix <float, 4, 1> CentroidVector;
		pcl::compute3DCentroid(*cloud_cluster2,  CentroidVector);
		//std::cout <<"Centroid for Cluster"<< j <<" is at:\n X:"<<
CentroidVector[0] << "\n Y:"<< CentroidVector[1]<< "\n Z:" <<CentroidVector[2]<<" " <<
std::endl;
		Eigen::Matrix3f CovarianceMatrix;
		pcl::computeCovarianceMatrixNormalized(*cloud_cluster2, CentroidVector,
CovarianceMatrix);
		Eigen::Vector2f Eigenvalues;
		Eigen::Matrix2f Eigenmatrix;
		Eigen::Matrix2f XYCovarianceMatrix = CovarianceMatrix.block<2,2>(0,0);

		pcl::eigen22(XYCovarianceMatrix,Eigenmatrix, Eigenvalues);
		double EigRatio;
		double EigShort;
		double EigLong;
		int N=3;
		EigShort = 2*N*sqrt(Eigenvalues(0));
		EigLong = 2*N*sqrt(Eigenvalues(1));
		EigRatio = EigLong/EigShort;

		std::cout<<"EigRatio = " << EigRatio<<" EigShort = "<<EigShort<<endl;

		if (EigRatio> 2 && EigShort>2.9 )
		{
			IdentifiedDockFinal += *cloud_cluster2;
		}

	}
}
pcl::io::savePCDFile("Plane Outliers1.pcd", CloudOutliers);
std::cout<<"There are "<< counter<<" datapoints within the bounds"<<endl;
//std::cout<< InlierCentroids<<std::endl;
std::cout<< "There are "<< j << " clusters that match the parameters\n"<<std::endl;
```

```
std::cout<<"There are " << n << " identified pilings"<<endl;
//std::cout<< "Pi = "<< Pi << "\n " <<std::endl;
textfile.close();
//return 0;




// Below controls the point cloud viewer
pcl::io::savePCDFile("Clusters1.pcd", cluster_cloud1);
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer2;
viewer= simpleVis(Cloud.makeShared(), IdentifiedDockFinal.makeShared());
while (!viewer->wasStopped ())
		{
			viewer->spinOnce (100);
			boost::this_thread::sleep (boost::posix_time::microseconds
(100000));
		}
viewer2= simpleVis(Cloud.makeShared());
while (!viewer2->wasStopped ())
		{
			viewer2->spinOnce (100);
			boost::this_thread::sleep (boost::posix_time::microseconds
(100000));
		}
}
```